UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.
09/204,973	12/03/98	EHNEBUSKE	D ATS-98-266

EXAMINER

TM21/1030

DUKE W YEE
CARSTENS YEE AND CAHOON
P O BOX 802334
DALLAS TX 75380

ART UNIT	PAPER NUMBER
2122	4

DATE MAILED: 10/30/01

Please find below and/or attached an Office communication concerning this application or proceeding.

Commissioner of Patents and Trademarks

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office

Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

SERIAL NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTORNEY DOCKET NO
09/209,973	12/3/98	D.L. Ehnebuske et al.	AT9-98-266

EXAMINER	
Ingberg	
ART UNIT	PAPER NUMBER
2122	5

DATE MAILED:

Please find below a communication from the EXAMINER in charge of this application.

Commissioner of Patents and Trademarks

Responsive to Communication Filed _____.

The enclosed is a correct copy of a reference relating to the last Office action. The correction is indicated below.

THE PERIOD FOR RESPONSE OF 3 MONTHS SET IN SAID OFFICE ACTION IS RESTARTED TO BEGIN WITH THE DATE OF THIS LETTER.

☐ Part 1 - Correct Reference Citation

_____

_____

by _____
Examiner

☒ Part 2 - Correct Reference Furnished:

_____

_____

by _____
Reference Order Center

KEVIN J. TESKA
SUPERVISORY
PATENT EXAMINER

enc. Rational Rose 4.0 chapter 10
Principles of Object Oriented Analysis & Design pages 261-289
Workflow Template - Using the WFT Development Environment
chapters 8 and 9 and Appendix

PTOL-316 (rev. 6/78)

Office Action Summary

Application No.	Applicant(s)
09/204,973	David Lars Ehnebuske et al.
Examiner	**Art Unit**
Todd Ingberg	2122

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136 (a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133).
- Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1) ☒ Responsive to communication(s) filed on *Feb 5, 1999*.

2a) ☐ This action is **FINAL.** 2b) ☒ This action is non-final.

3) ☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11; 453 O.G. 213.

Disposition of Claims

4) ☒ Claim(s) *1-98* is/are pending in the application.

4a) Of the above, claim(s) _____ is/are withdrawn from consideration.

5) ☐ Claim(s) _____ is/are allowed.

6) ☒ Claim(s) *1-98* is/are rejected.

7) ☐ Claim(s) _____ is/are objected to.

8) ☐ Claims _____ are subject to restriction and/or election requirement.

Application Papers

9) ☐ The specification is objected to by the Examiner.

10) ☐ The drawing(s) filed on _____ is/are objected to by the Examiner.

11) ☒ The proposed drawing correction filed on *Mar 15, 1999* is: a) ☒ approved b) ☐ disapproved.

12) ☐ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. § 119

13) ☐ Acknowledgement is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d).

a) ☐ All b) ☐ Some* c) ☐ None of:

1. ☐ Certified copies of the priority documents have been received.
2. ☐ Certified copies of the priority documents have been received in Application No. _____.
3. ☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

*See the attached detailed Office action for a list of the certified copies not received.

14) ☐ Acknowledgement is made of a claim for domestic priority under 35 U.S.C. § 119(e).

Attachment(s)

15) ☒ Notice of References Cited (PTO-892)
16) ☒ Notice of Draftsperson's Patent Drawing Review (PTO-948)
17) ☐ Information Disclosure Statement(s) (PTO-1449) Paper No(s). _____
18) ☒ Interview Summary (PTO-413) Paper No(s). 4
19) ☐ Notice of Informal Patent Application (PTO-152)
20) ☐ Other:

DETAILED ACTION

Claims 1 - 98 have been examined.

Claims 12, 18, 19, 20 , 21, 30, 54, 70 71, 72, 76, 77, 78,80 and 81 were amended in a preliminary amendment received February 5, 1999.

Drawings

1. The corrected or substitute drawings were received on March 15, 1999. These drawings are approved by the Draft's Person.

Information Disclosure Statement

2. In the event the invention is related to the Assignee's (IBM) product **FLOWMARK** (IBM Trademark #2006543) the Applicant is reminded of their duty to disclose. FLOWMARK dates back to filing for Trademark on December 16, 1993. Date of first use in commerce June 28, 1995.

3. The Applicant makes specific reference to Object Management Group (OMG), the Assignee (IBM) is a long time member of this standards organization for Object technology. In the event work of OMG is relevant to Common Object Request Broker (**CORBA**), application frameworks and Business Objects, the Applicant is reminded of their duty to disclose.

4. In the event, the invention is related to the tool **ISMOD** used by IBM in 1990 to model dimensions of business such as "data and information, data criticism, data flow identification, data flow metrics, data qualifiers, event triggers, location, organization, processes", as described in

IBM System Journal, Vol 29 Issue 4, page 509, page 17, then, the Applicant is reminded of their duty to disclose.

5. In the event, the invention is related to the tool **DevelopMate** used by IBM in 1990 to develop a business/enterprise model with dimensions that include business processes, business data events...", as described in IBM System Journal, Vol 29 Issue 4, page 509, page 17, then, the Applicant is reminded of their duty to disclose.

6. In the event, the invention is related to the product of **Newi** from Integrated Objects - a joint venture of IBM and Softwright based in England, the Applicant is reminded of their duty to disclose.

7. In the event, the invention is related to feature "rule editor probe", contained in the product **Object Management Workbench** (OMW) of Intellicorp, the Applicant is reminded of their duty to disclose.

Currently, compliance with this request is only covered under 1.56 but could be the subject of a future Requirement For Information (RFI) under 1.105.

Specification

8. The Specification contains and error on page 6. No case with the Serial number of 09/993,718 has been filed with the USPTO at this time. The correct serial number is needed as part of an amendment.

Examiner's Interpretation

9. The following are Examiner interpretations.

a. ***Modeling*** - The Martin book teaches the use of modeling an enterprise (**Martin**, page 247 - 249) operation and provides Appendix A, Recommended Diagraming Standards. Martin page 285 illustrates the transition from the problem domain, to modeling, to OO Design to code. The Martin reference has many chapters covering modeling and discusses the models are tied together to generate code (**Martin,** page 155, box).

The Martin reference also provides some examples. These models are not viewed as static. The Martin reference teaches modeling the enterprise. The principles and techniques are dynamic.

b. ***Flow Control*** - The Applicant does not explicitly claim flow control. However, when the Applicant states in the claims "method logic is continuous" this is interpreted as meaning the method (a common feature in object technology) can run until a outside interrupt occurs. This is a product of flow control resulting from the logic structure of a computer program. There are many claims to what the Examiner interprets as claims to flow control. Flow control is the tracing the path of an executing of a program. The exact path the execution of a program will follow is determined by the values of the attributes and the control conditions encountered. The examples of the programming constructs such as Martin page 148 show the difference paths flow control can take depending on execution of operations such as CHECK IN COPY , FILL REQUEST and BOOK OVERDUE. The values of attributes are tested to determine the path taken. Many claims have made claim to flow control which is inherent to the execution of a computer programs.

c. Claim 3 contains the following limitation ".... the step of defining a first control point further comprises:

(a1) decorating the object to dynamically insert a first control point such that the object acquires this new control point."

The Examiner interprets the "decorating" to mean the entry of programming information such as the operations/methods and the entry of control points in a programming environment.

Claim Rejections - 35 USC § 102

10. The following is a quotation of the appropriate paragraphs of 35 U.S.C.§ 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless --
>
> (a) the invention was known or used by others in this country, or patented or described in a printed publication in this or a foreign country, before the invention thereof by the applicant for a patent.
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

11. Claims 1 - 98 are rejected under 35 U.S.C. 102(a) as being anticipated by "Principles of Object-Oriented Analysis and Design", James **Martin**, published June 1, 1992.

The Martin teaches the underlying theory of building an Object Oriented Computer Aided Software Engineering (OO-CASE) tools in his 1992 text book. The Martin should be taken as a whole, however, focus of the rejection is on the Chapters 9 and 10. The Martin references covers the very basics of object technology that one of ordinary skill should have known well before the time of invention:

Chapter 2 - Basic Concepts

Chapter 3 - Why Object-Oriented ?

Chapter 4 - Basic Guidelines

Chapter 6 - Categorizing Objects

Chapter 7 - Relationships Among Object Types

Chapter 8 - State and State Changes

Chapter 9 - Events, Triggers, and Operations

Chapter 10 - Rules

Chapter 11 - How Diagrams Interrelate

Chapter 12 - Basic Concepts of OO Design

Chapter 15 - Method Creation

Chapter 18 - OO-CASE Tools

Appendix A - Recommended Diagraming Standards

The Martin book in addition to containing foundation knowledge of object oriented technology it teaches applying a set of rules comprising the placing of logic (program statements) in a pre-method control before the logic of a method and post method control point after the logic of a method. Martin also teaches associating a set of rules with each control point based on the class of the object in which the method resides, name of the method and type of control point and invoking methods.

Claim 1

Martin anticipates a computer implemented process for applying a set of rules (**Martin,** Chapter 10, RULES, and page 138-139 and 249-251), the process comprising:

(a) placing a pre-method control before logic of a method (**Martin,** page 142, operation precondition) and post-method control point after the logic of the method (**Martin,** page 142, postcondition)

(b) associating a set rules with each control point (**Martin,** page 142, 147 "Operation" as per (a) above) based on a class of object in which the method resides (**Martin,** page 143, "... rules associated with diagrams of OO ...") , name of the method, and type of control point, whether the pre-method control point or the post-method control point (**Martin,** page 142, operation precondition) ;

(c) invoking the method (**Martin**, page 116), wherein encountering each control point during the execution of the method comprises (**Martin,** page 142, postcondition):

(i) determining if the encountered control point is active (**Martin,** page 122, IF structure in center diagram) ;

(ii) on the basis of an active control point (Interpreted as the result of the IF structure above further described in Appendix A on page 381 Control Conditions):

1) selecting rules based on a set of rules associated with the active control point associated in step (**Martin,** page 122, first diagram example is the control condition to fire missile) (b);

2) running the selected rules (**Martin,** page 122, rule that lead to the control condition);

3) obtaining results from running the rules (**Martin,** page 122, trigger rule at the bottom of the page); and

4) combining the results using a combining algorithm specified by the control point (**Martin,** page 122, A control condition can function as a combining algorithm as seen in diagram in middle of the page and page 126 Figure 9.9 and **Martin** teaches a way to have a combing algorithm where one of three operations are selected as on page 124, and **Martin** teaches a way to have a combining algorithm where one can be selected as taught in the mutually exclusive notation on the bottom of page 125).

Claim 2

Martin anticipates a computer implemented process for applying a set of rules (**Martin**, Page 166, Rules Linked to Diagrams a product and **Martin**, page 172 from operations to methods) comprising:

(a) defining an object (**Martin,** page 171, CLASS - bottom of page);

(b) defining at least one method in the object (**Martin**, page 173, Method - top of page and page 116 and page 167 Rule editor);

(c) defining a control point just before logic of at least one method (**Martin**, page 173, diagram in center of page); and

(d) associating a set of rules with the control point (**Martin**, page 173, diagram in center of page).

Claim 3

In the process of claim 2, the step of defining a first control point further comprises:

(a1) decorating the object to dynamically insert a first control point such that the object acquires this new control point (**Martin,** teaches how the object model and rules are associated in Chapter 5 page 59, on page 60 of Martin last paragraph OO-CASE tools states last paragraph "Every time we tell an advanced CASE tool about classes, inheritance, and so on, it should generate code." The next sentence mentions code can be generated from rules. The reference also mentions the diagrams are executable such as on page 142 - The reference teaches Object Oriented Computer Aided Software Engineering (OO-CASE) tools and their features. OO-CASE tools as described create objects and methods much of which is described in claim 2).

Claim 4

In the process of claim 2, the step of defining at least one control point further comprises:

(c1) adding the at least one control point through the technique of generating required code in the compiler or with a preprocessor. As per claim 3 the diagrams generate code also see the definition of Instant CASE or I-CASE **Martin,** pages 7, 243, 282-283, 284, 293, 294, 351 and 352.

Claim 5

In the process of claim 2, the step of defining at least one control point further comprises:

(c1) manually inserting the at least one control point and encoding the control point in the implementation of a hosting object. As per claim 3.

Claim 6

In the process of claim 2, the step of defining at least one control point further comprises:

(cl) externalizing the at least one control point as a class and instantiating it at the at least one control point (**Martin**, page 133 - 136, BOX and claim 1 and page 167 OMW screen shot)

Claim 7

The process of claim 2 further comprises:

(e) defining a second control point just after the logic of each method; and

(f) associating a second set of rules with the second control point (**Martin,** in many locations teaches the ability to have more than 1 control point and additional rules see page 164 for an example).

Claim 8

In the process of claim 7, wherein the rules in the second set of rules are associated to the second control point without considering the rules in the first set of rules associated with the at least one control point as per claim 7.

Claim 9

In the process of claim 7, wherein a set of rules is defined as having N number of rules, N being at least zero. As per claim 1 the presents of a single RULE means it exists and the count of rules present is greater than zero.

Claim 10

In the process of claim 2, the step of associating at least one control point further comprises:

(cl) defining, with a control point, at least one of a rule selecting algorithm and a rule-results combination algorithm As per claim 1.

Claim 11

The process of claim 2, further comprises:

(e) changing rules associated with the control point contained in the set of rules. As per claims 3 and claim 4 OO-CASE and I-CASE by definition.

Claim 12

Martin anticipates a computer implemented process for applying a set of rules (as per claim 2), comprising:

(a) invoking a method in an object (as per claim 2);

(b) encountering an active control point during the invocation of the method (as per claim 2);

(c) selecting rules associated with the method of the object at the control point (as per claim 2);

(d) invoking the rules (as per claim 2); and

(e) combining results from invoking the rules as per claim 1.

Claim 13

The process of claim 12, wherein the rules perform a variety of actions (Martin, page 164, a variety of actions can occur such as Invoice Student OR Get Dorm depending on the outcome of the Remote Student registered condition) conditioned by the fact that rules may be associated with particular, regularly occurring points in the object model Martin,

(Martin, page 166, RULES LINKED TO DIAGRAMS, "The importance of rules was emphasized in Chapter 10 which indicated that rules can be connected to any of the OO diagrams".)

Claim 14

The process of claim 12, wherein the rules perform at least one function which varies over time (**Martin,** page 117, clock events and page 144 Rules Associated with Event Diagrams - " If time is between 9 AM to 5 PM", **Martin**, page 394, Clock Events).

Claim 15

A process of claim 12, wherein a control point occurs just before logic of the method begins, just after the logic of the method completes, or at both just before logic of the method begins and just after the logic of the method completes as per claim 1.

Claim 16

Martin anticipates a computer implemented process for applying a set of rules (as per claim 2) comprising:

(a) defining an object (as per claim 2);

(b) defining at least one method in the object (as per claim 2);

(c) defining at least one control point in the at least one method (as per claim 2).

(d) defining rules to the at least one control point on basis the object's class name, method, name, and position of the at least one control point in the method (Martin, Chapter 12, BASIC CONCEPTS OF OO DESIGN, page 172 - 173, the relationship between classes and objects and the relationship between rules and object modeling).

Claim 17

In the process of claim 16, further comprising the step of activating at least one control point having associated rules as per claim 1.

Claim 18

The process of claim 16 further comprises:

(e) encountering a first control point (**Martin,** page 173, control point with a Time Event) ;

(f) running the rules associated with the first control point (**Martin,** page 173, control point with a Time Event); and

(g) affecting behavior of the object based on running the rules associated with the first control point (The flow control is controlled by the Rule associated with the Control point as per **Martin,** page 381).

Claim 19

In the process of claim 18, the step of affecting the behavior of the object further comprises:

(h) associating different rules to a control point (as per claim 14 - Different rules based on the time of day affects the flow control/ behavior).

Claim 20

In the process of claim 18, the step of affecting the behavior of the object further comprises:

(h) defining another control point (Examiner Interpretation of "defining another control point" the meaning could be at design time or runtime. Design time would involve the interaction with the OO-CASE tool as on **Martin**, page 162, Run time would mean the behavior changes value such as attribute which influence the path the control flow takes. This is the point of programming. The

ability to model a problem domain and execute code that process information that reflects the modeled problem - Flow Control as determined by the running of the program such as **Martin,** page 163).

Claim 21

In the process of claim 18, the step of modifying the object further comprises:

(h) associating rules to a second control point (**Martin,** page 163, Multiple control points defined).

Claim 22

In the process of claim 16, further comprising a step of deactivating the at least one control point.(As per claim 1. The control point is determined if it is active or not. If one takes the **Martin**, page 163 example where the timed event is part of the Waitlisted functionality the Timed event occurs at a specific time the Timed Event is one example of activating and deactivating the control point also see **Martin,** Appendix A, page 394)

Claim 23

Martin anticipates a computer implemented process for applying a set off rules (as per claim 2), comprising

(a) defining an object (as per claim 2);

(b) defining a method in the object (as per claim 2);

(c) defining a first control point of the method (as per claim 2);

(d) determining rules associated with the first control point (as per claim 2);

(e) defining a second control point of the method (as per claim 2); and

(f) determining rules associated with the second control point (as per claim 2).

Claim 24

A computer implemented process as in claim 23 further comprising:

(g) separately selecting, running and combining the results of rules determined to be associated with either control point as per claim 1.

Claim 25

In the process of claim 23 wherein the first control point is a pre-method trigger point (**Martin**, page 142, diagram top of page, page 381 Trigger Rules).

Claim 26

In the process of claim 23 wherein the second control point is a post-method trigger point (**Martin**, page 115, Postconditions in cause and effect isolation, page 141, Post Condition page 381, Trigger Rule).

Claim 27

Martin anticipates a computer implemented process for defining an object (Martin, page 166 - 167, Link between, Diagrams, Rules and Objects) comprising:

defining an object; (**Martin**, page 144, Box 10.3, and page 169 - 176 and as per claim 2)

defining a method in the object by: defining method logic (as per claim 2) ;

placing the method logic in the method (**Martin**, page 173, methods is the Specification of an operation and as per claim 2);

defining at least one control point (as per claim 2);

and placing the at least one control point in the method wherein the method logic is continuous. (**Martin**, page 224, DO and FOR loops, page 225, Loops in action diagrams).

Claim 28

A computer implemented process for defining an object as in claim 27, wherein the step of placing the at least one control point further comprises placing the at least one control in the method before the method logic (as per claim 1).

Claim 29

A computer implemented process for defining an object as in claim 27, wherein the step of placing the at least one control point further comprises placing the at least one control in the method after the method logic (as per claim 1).

Claim 30

A computer implemented process for defining an object as in claim 27, wherein the at least one control point comprises two control points and further comprises: placing a first control point in the method before the method logic; and placing a second control point in the method after the method logic (Martin, page 126, Figure 9.9).

Claim 31

A computer implemented process for defining an object as in claim 27, further comprises: flagging the at least one control point on the basis of being active (as per claim 1).

Claim 32

A computer implemented process for defining an object as in claim 27, wherein the step of defining the at least one control point further comprising: defining a rule selection algorithm associated with the at least one control point (**Martin,** page 168, control point rule illustrated).

Claim 33

A computer implemented process for defining an object as in claim 27, wherein the step of defining the at least one control point further comprising: defining a rule result combination algorithm associated with the at least one control point. As per claim 1.

Claim 34

A computer implemented process for defining an object as in claim 27, wherein the step of defining the at least one control point further comprises: defining a rule selection algorithm for the at least one control point; and defining a rule result combination algorithm for the at least one control point As per claim 1.

Claim 35

A computer implemented process for defining an object as in claim 27, further comprising: associating at least one rule with the at least one control point. As per claim 32.

Claim 36

Martin anticipates a computer implemented process for defining a rule comprising: creating the rule (**Martin**, page 167, Rule Editor) ; associating the rule with an object class (**Martin**, page 167, Figure 11.14); associating the rule with a method within the object class (Martin, page 173,

operations are methods); and associating the rule with an occurrence of a control point within the method (**Martin,** page 168, Figure 11.16).

Claim 37

A computer implemented process for defining a rule as in claim 36 wherein the occurrence of the control point within the method being before method logic. As per claim 1.

Claim 38

A computer implemented process for defining a rule as in claim 36 wherein the occurrence of control point within the method being after method logic. As per claim 1.

Claim 39

A computer implemented process for defining a rule as in claim 36, further comprising: associating the rule with another object class (**Martin,** page 267, the ability to access a method/Rule from more than one object and the concept of Reuse which is a key factor in object oriented technology **Martin**, page 248, Box 16.2 Maximize reusability) (This claim could also be interpreted as claiming the principle of inheritance as described on **Martin**, page 266 - 268).

Claim 40

A computer implemented process for defining a rule as in claim 36, further comprising: associating the rule with another method within the object class. As per claim 39.

Claim 41

A computer implemented process for defining a rule as in claim 36, further comprising: associating the rule with another control point within the method of the object class (**Martin,** page 166 - 168, the rule associated to the control point , page 233, RULES)

Claim 42

Martin anticipates a computer implemented process for applying a set of rules (as per claim 2), comprising: selecting an object class; selecting a method within the object class; invoking the method; processing rules associated with the method comprising: encountering a control point associated with the method; determining if the control point is active; and finding at least one rule associated with an active control point. (Interpreted as the running of the code generated by claim 2).

Claim 43

A computer implemented process for applying a set of rules as in claim 42, wherein the step of finding at least one rule further comprises: accessing a selecting algorithm associated with the active control point (as per claim 1); and selecting at least one rule using the selecting algorithm (as per claim 10 and The IF structure in the control point as per **Martin**, page 168).

Claim 44

A computer implemented process for applying a set of rules as in claim 42, where in the step of processing rules further comprises: running the at least one rule; determining results from running the at least one rule; accessing a combining algorithm associated with the control point; and combining the results using the combining algorithm. As per claim 1.

Claim 45

Martin anticipates a computer implemented process for applying a set of rules, comprising: selecting an object class; selecting a method within the object class; invoking the method; processing rules comprising: encountering a control point; accessing a selecting algorithm associated with the control point; and selecting at least one rule using the selecting algorithm. As per claim 42.

Claim 46

Martin anticipates a computer implemented process for applying a set of rules, comprising: selecting an object class; selecting a method within the object class; invoking the method; processing rules comprising: encountering a control point; finding at least one rule associated with the control point; running the at least one rule; determining results on the basis of running the at least one rule; accessing a combining algorithm associated with the control point; and combining the results using the combining algorithm. As per claim 1 - the running of the executable generated from the model.

Claim 47

Martin anticipates a computer implemented process for applying a set of rules, comprising: selecting an object class; selecting a method within the object class; invoking the method; processing rules comprising: encountering a first control point associated with the method; determining if the first control point is active (the running of code from claims 1 and 2 and implementations such as page 164 Fig 11.10); executing method logic of the method;

encountering a second control point associated with the method; determining if the second control point is active; finding a set of rules associated with one of the first control point and the second control point, wherein the set of rules contains not less than zero rules as per claim 9.

Claim 48

Martin anticipates a computer implemented process for applying a set of rules, comprising: selecting an object class; selecting a method within the object class; invoking the method; processing rules comprising: encountering a control point associated with the method; finding at least one rule associated with the control point prior to executing method logic of the method; running the at least one rule; obtaining results on the basis of running the at least one rule; and controlling the method on the basis of the results (The running of code from claims 1 and 2).

Claim 49

A computer implemented process for applying a set of rules as in claim 48, wherein the step of controlling the method comprises: exiting the method. **Martin**, page 236, use of "return" in C++ and it is well known in C++ that reaching the end of a method such as flow control reaching the last "}" in the method declassify will return flow control to the method that called this method or terminate. In either path the method has performed an exit.

Claim 50

A computer implemented process for applying a set of rules as in claim 48, wherein the step of controlling the method comprises: executing method logic of the method. The running of the executable code produced by the modeling from claim 1 - Flow control.

Claim 51

Martin anticipates a data processing system for defining an object comprising: defining means for defining an object; defining means for defining a method in the object by: defining means for defining method logic; placing means for placing the method logic in the method; defining means for defining at least one control point; and placing means for placing the at least one control point in the method wherein the method logic is continuous. As per claim 27.

Claim 52

A data processing system for defining an object as in claim 51, wherein the step of placing the at least one control point further comprises placing means for placing the at least one control in the method before the method logic. As per claim 1.

Claim 53

A data processing system for defining an object as in claim 51, wherein the step of placing the at least one control point further comprises placing means for placing the at least one control in the method after the method logic. As per claim 1.

Claim 54

A data processing system for defining an object as in claim 51, wherein the at least one control point comprises two control points and further comprises: placing means for placing a first control point in the method before the method logic; and placing means for placing a second control point in the method after the method logic. As per claims 2 and 7.

Claim 55

A data processing system for defining an object as in claim 51, further comprises: flagging means for flagging the at least one control point on the basis of being active. As per claim 31.

Claim 56

A data processing system for defining an object as in claim 51, wherein the step of defining the at least one control point further comprising: defining means for defining a rule selection algorithm associated with the at least one control point. As per claim 32.

Claim 57

A data processing system for defining an object as in claim 51, wherein the step of defining the at least one control point further comprising: defining means for defining a rule result combination algorithm associated with the at least one control point as per claim 10.

Claim 58

A data processing system for defining an object as in claim 51, wherein the step of defining the at least one control point (as per claims 1 and 2) further comprises: defining means for defining a rule selection algorithm for the at least one control point; and defining a rule result combination algorithm for the at least one control point. As per claim 34.

Claim 59

A data processing system for defining an object as in claim 51, further comprising: associating means for associating at least one rule with the at least one control point. As per claim 8.

Claim 60

Martin anticipates a data processing system for defining a rule comprising: creating means for creating the rule; associating means for associating the rule with an object class; associating means for associating the rule with a method within the object class; and associating means for associating the rule with an occurrence of a control point within the method. As per cliam 36.

Claim 61

A data processing system for defining a rule as in claim 60 wherein the occurrence of the control point within the method being before method logic. As per claim 1.

Claim 62

A data processing system for defining a rule as in claim 60 wherein the occurrence of control point within the method being after method logic. As per claim 1.

Claim 63

A data processing system for defining a rule as in claim 60, further comprising: associating means for associating the rule with another object class. (**Martin,** page 267, the ability to access a method/Rule from more than one object and the concept of Reuse which is a key factor in object oriented technology **Martin**, page 248, Box 16.2 Maximize reusability) (This claim could also be interpreted as claiming the principle of inheritance as described on **Martin**, page 266 - 268).

Claim 64

A data processing system for defining a rule as in claim 60, further comprising: associating means for associating the rule with another method within the object class. As per claim 39.

Claim 65

A data processing system for defining a rule as in claim 60, further comprising: associating means for associating the rule with another control point within the method of the object class. As per claim 1.

Claim 66

Martin anticipates a data processing system for applying a set of rules, comprising: selecting means for selecting an object class; selecting means for selecting a method within the object class; invoking means for invoking the method; processing means for processing rules associated with the method comprising: encountering means for encountering a control point associated with the method; determining means for determining if the control point is active; and finding means for finding at least one rule associated with an active control point. As per claim 42.

Claim 67

A data processing system for applying a set of rules as in claim 66, wherein the step of finding at least one rule further comprises: accessing means for accessing a selecting algorithm associated with the active control point; and selecting means for selecting at least one rule using the selecting algorithm. As per claim 43.

Claim 68

A data processing system for applying a set of rules as in claim 66, where in the step of processing rules further comprises: running means for running the at least one rule; determining means for determining results from running the at least one rule; accessing means for accessing a combining

algorithm associated with the control point; and combining means for combining the results using the combining algorithm. As per claim 44.

Claim 69

Martin anticipates a data processing system for applying a set of rules, comprising: selecting means for selecting an object class; selecting means for selecting a method within the object class; invoking means for invoking the method; processing means for processing rules comprising: encountering means for encountering a control point; accessing means for accessing a selecting algorithm associated with the control point; and selecting means for selecting at least one rule using the selecting algorithm. As per claim 45.

Claim 70

Martin anticipates a data processing system for applying a set of rules, comprising: selecting means for selecting an object class; selecting means for selecting a method within the object class; invoking means for invoking the method; processing means for processing rules comprising: encountering means for encountering a control point; finding means for finding at least one rule associated with the control point; running means for running the at least one rule; determining means for determining results on the basis of running the at least one rule; accessing means for accessing a combining algorithm associated with the control point; and combining means for combining the results using the combining algorithm. As per claim 46.

Claim 71

Martin anticipates a data processing system for applying a set of rules, comprising: selecting means for selecting an object class; selecting means for selecting a method within the object class; invoking means for invoking the method; processing means for processing rules comprising: encountering means for encountering a first control point associated with the method; determining means for determining if the first control point is active (as per claim 2); executing means for executing method logic of the method (as per claim 2); encountering means for encountering a second control point associated with the method; determining means for determining if the second control point is active; finding, means for finding a set of rules associated with one of the first control point and the second control point (as per claim 7), wherein the set of rules contains not less than zero rules. As per claim 9.

Claim 72

Martin anticipates a data processing system for applying a set of rules, comprising: selecting means for selecting an object class; selecting means for selecting a method within the object class; invoking means for invoking the method; processing means for processing rules comprising: encountering means for encountering a control point associated with the method; finding means for finding at least one rule associated with the control point prior to executing method logic of the method; running means for running the at least one rule; obtaining means for obtaining results on the basis of running the at least one rule; and controlling means for controlling the method on the basis of the results. As per claim 48.

Claim 73

A data processing system for applying a set of rules as in claim 72, wherein the step of controlling the method comprises: exiting means for exiting the method. As per claim 49.

Claim 74

A data processing system for applying a set of rules as in claim 72, wherein the step of controlling the method comprises: executing means for executing method logic of the method. As per claim 50.

Claim 75

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for defining an object, the instruction comprising: instructions for defining an object; instructions for defining a method in the object by: instructions for defining method logic; instructions for placing the method logic in the method; instructions for defining at least one control point; and instructions for placing the at least one control point in the method wherein the method logic is continuous. As per claim 51.

Claim 76

A computer program product for defining an object as in claim 75, wherein the instruction of placing the at least one control point further comprises placing the at least one control point in the method before the method logic. As per claim 1.

Claim 77

A computer program product for defining an object as in claim 75, wherein the instruction of placing the at least one control point further comprises placing the at least one control point in the method after the method logic. As per claim 1.

Claim 78

A computer program product for defining an object as in claim 75, wherein the at least one control point further comprises two control points and further comprises: instructions for placing a first control point in the method before the method logic; and instructions for placing a second control point in the method after the method logic.As per claim 1.

Claim 79

A computer program product for defining an object as in claim 75, further comprises: instructions for flagging the at least one control point on the basis of being active. As per claim 31.

Claim 80

A computer program product for defining an object as in claim 75, wherein the instruction of defining the at least one control point further comprising: instructions for defining a rule selection algorithm associated with the at least one control point. As per claim 32.

Claim 81

A computer product for defining an object as in claim 75, wherein the instruction of defining the at least one control point further comprises: instructions for defining a rule combination algorithm associated with the at least one control point. As per claim 33.

Claim 82

A computer program product for defining an object as in claim 75, wherein the step of defining the at least one control point further comprises: instructions for defining a rule selection algorithm for the at least one control point; and instructions for defining a rule result combination algorithm for the at least one control point.As per claim 34.

Claim 83

A computer program product for defining an object as in claim 75, further comprising: instructions for associating at least one rule with the at least one control point. As per claim 35.

Claim 84

Martin anticipates a computer program product embodied on a computer" readable medium containing instructions for a computer implemented process for defining a rule, the instruction comprising: instructions for creating the rule; instructions for associating the rule with an object class; instructions for associating the rule with a method within the object class; and instructions for associating the rule with an occurrence of a control point within the method. As per claim 36.

Claim 85

A computer program product for defining a rule as in claim 84 wherein the occurrence of the control point within the method being before method logic. As per claim 1.

Claim 86

A computer program product for defining a rule as in claim 84 wherein the occurrence of control point within the method being after method logic.As per claim 1.

Claim 87

A computer program product for defining a rule as in claim 84, further comprising: instructions for associating the rule with another object class. As per claim 39 or 63.

Claim 88

A computer program product for defining a rule as in claim 84, further comprising: instructions for associating the rule with another method within the object class. As per claim 39 or 64.

Claim 89

A computer implemented process for defining a rule as in claim 84, further comprising: instructions for associating the rule with another control point within the method of the object class. As per claim 65.

Claim 90

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for applying a set of rules, the instruction comprising: instructions for selecting an object class; instructions for selecting a method within the object class; instructions for invoking the method; instructions for processing rules associated with the method comprising: instructions for encountering a control point associated with the method; instructions for determining if the control point is active; and instructions for finding at least one rule associated with an active control point. As per claim 1.

Claim 91

A computer program product for applying a set of rules as in claim 90, wherein the step of finding at least one rule further comprises: instructions for accessing a selecting algorithm associated with

the active control point; and instructions for selecting at least one rule using the selecting algorithm. As per claim 43.

Claim 92

A computer program product for applying a set of rules as in claim 90, where in the step of processing rules further comprises: instructions for running the at least one rule; instructions for determining results from running the at least one rule; instructions for accessing a combining algorithm associated with the control point; and instructions for combining the results using the combining algorithm. As per claim 1.

Claim 93

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for applying a set of rules, the instruction comprising: instructions for selecting an object class; instructions for selecting a method within the object class; instructions for invoking the method; instructions for processing rules comprising: instructions for encountering a control point; instructions for accessing a selecting algorithm associated with the control point; and instructions for selecting at least one rule using the selecting algorithm. As per claim 42.

Claim 94

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for applying a set of rules, the instruction comprising: instructions for selecting an object class; instructions for selecting a

method within the object class; instructions for invoking the method; instructions for processing rules comprising: instructions for encountering a control point; instructions for finding at least one rule associated with the control point; instructions for running the at least one rule; instructions for determining results on the basis of running the at least one rule; instructions for accessing a combining algorithm associated with the control point; and instructions for combining the results using the combining algorithm. As per claim 1 or 46.

Claim 95

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for applying a set of rules, the instruction comprising: instructions for selecting an object class (as per claim 42); instructions for selecting a method within the object class; instructions for invoking the method; instructions for processing rules comprising: instructions for encountering a first control point associated with the method; instructions for determining if the first control point is active; instructions for executing method logic of the method; instructions for encountering a second control point associated with the method; instructions for determining if the second control point is active (Asper claim 47); instructions for finding a set of rules associated with one of the first control point and the second control point, wherein the set of rules contains not less than zero rules. As per claim 9.

Claim 96

Martin anticipates a computer program product embodied on a computer readable medium containing instructions for a computer implemented process for applying a set of rules, the

instruction comprising: instructions for selecting an object class; instructions for selecting a method within the object class; instructions for invoking the method; processing rules comprising: instructions for encountering a control point associated with the method; instructions for finding at least one rule associated with the control point prior to executing method logic of the method; instructions for running the at least one rule; instructions for obtaining results on the basis of running the at least one rule; and instructions for controlling the method on the basis of the results. As per claim 48.

Claim 97

A computer program product for applying a set of rules as in claim 96, wherein the step of controlling the method comprises: instructions for exiting the method. **Martin**, page 236, use of "return" in C++ and it is well known in C++ that reaching the end of a method such as flow control reaching the last "}" in the method declassify will return flow control to the method that called this method or terminate. In either path the method has performed an exit.

Claim 98

A computer program product for applying a set of rules as in claim 96, wherein the step of, controlling the method comprises: instructions for executing method logic of the method. As per claims 50.

Summary

12. The rejection under Martin is a fundamental teaching from June 1992 which contains features inherent to Object Oriented technology and a primer book on the technology. The WFT

rejection is a commercial product for use and for sale since April 1997 which implementations Rules in an Object Oriented CASE tool. Both rejections support the concept that the tools are designed for the end user to implement rules without formal programming experience. Rational Rose is also a OO-CASE tool which teaches the ability implement rules at a Programmer level.

Conclusion

13. The undisclosed prior art of Assignee (**IBM**) is made of record and not relied upon is considered pertinent to applicant's disclosure.

A. "Business/Enterprise Modeling", Robert Katz, **IBM Systems Journal**, Armonk 1990, Vol 29, Issue 4, page 509, 17 pages

B. "The Impact of Object-Orientation on Software Development", A.A.R. Cockburn, **IBM Systems Journal**, 1993, Vol 32, No 3, pages 420-444.

C. "Process Automation in Software Application Development", K.D. Saracelli et al, **IBM Systems Journal**, 1993, Vol 32, No 3, pages 376-396.

14. Prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

D. Template Software Inc., product line contains the SNAP programming language and the Workflow Template. The a subset of the documentation sets for the products contains the following manuals that cover their Workflow system with RULES.

Workflow released April 3, 1997

Developing a WFT Workflow System

Using the WFT Development Environment

E. Rational Corporation's product Rational Rose C++ version 4, released in 1996.

Rational Rose C++ version 4.0 contains a *document set* containing the following documents:

- Round-Trip Engineering with Rational Rose/C++
- Using Rational Rose 4.0
- UML, Booch & OMT Quick Reference for Rational Rose 4.0

This product set should be viewed as the state of the technology in the mid 1990's. The Rational Rose product was eliminated from consideration because the Applicant appears to be using the term "Externalizing Rules" to mean the domain expert is the end user not the programming. Rational Rose the rules are implemented by the programmer not the domain expert. Both Martin and Template implement rules which are intended to be entered by the domain expert.

Correspondence Information

15. Any inquiry concerning this communication or earlier communications from the Examiner should be directed to **Todd Ingberg** whose telephone number is **(703) 305-9775.** The Examiner can normally be reached on Monday, Tuesday, Thursday and Friday from 6:30 a.m. to 5:00 p.m. Until, October 22, 2001 then the Examiner's work schedule will be Monday through Thursday from 6:30 a.m. to 5:00 p.m.

If attempts to reach the examiner by telephone are unsuccessful, the **Examiner's Supervisor**, Mark Powell is on an extended work detail, **Acting Supervisor Kevin Teska** can be

reached at **(703)305-9704.** Any response to this office action should be mailed to: **Director of Patents and Trademarks Washington, D.C. 20231** or **faxed to: (703) 308-9051**, (for formal communications intended for entry) Or: **(703) 308-1396**, (for informal or draft communications, please label "PROPOSED" or "DRAFT") **Hand-delivered** responses should be brought to **Crystal Park II, 2121 Crystal Drive Arlington, Virginia, (Receptionist located on the sixth floor).**

Todd Ingberg

September 30, 2001

KEVIN J. TESKA
SUPERVISORY
PATENT EXAMINER

Notice of References Cited

Applicant/Patent	Application/Control No.	
David Lars Ehnebuske, et al.	09/204,973	
Examiner	Art Unit	
Todd Ingberg	2122	Page 1 of 4

U.S. PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Name	Classification[2]	
	A	5,930,512	7/1999	Boden et al.	717	1
	B	5,960,200	9/1999	Eager et al.	717	5
	C	6,074,431	6/2000	Watanabe et al.	717	2
	D	6,158,044	12/2000	Tibbets	717	1
	E	6,167,564	12/2000	Fonatana et al.	717	1
	F					
	G					
	H					
	I					
	J					
	K					
	L					
	M					

FOREIGN PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Country	Name	Classification[2]	
	N						
	O						
	P						
	Q						
	R						
	S						
	T						

NON-PATENT DOCUMENTS

		Include, as applicable: Author, Title, Date, Publisher, Edition or Volume, Pertinent Pages
	U	"Principles of Object Oriented Analysis and Design", James Martin, Chpaters 1 - 22, published June 1, 1992.
	V	Template Software Inc., "Using the WFT Development Environment", WFT Version 8.0 Chapters 1 - 9, released 1997
	W	Template Software Inc., "Developing a WFT Workflow System", WFT Version 8.0, Chapter 1 - 10, released 1997
	X	"Template Software Rolls Out Corporate and Product Growth Strategies at Solutions '97 Conference", PR Newswire, April 3, 1997.

* A copy of this reference is not being furnished with this Office action. See MPEP § 707.05(a). [1] Dates in MM-YYYY format are publication dates. [2] Classifications may be U.S. or foreign.

Notice of References Cited	Applicant/Patent: David Lars Ehnebuske et al.	Application/Control No.: 09/204,973
	Examiner: Todd Ingberg	Art Unit: 2122
		Page 2 of 4

U.S. PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Name	Classification[2]	
	A					
	B					
	C					
	D					
	E					
	F					
	G					
	H					
	I					
	J					
	K					
	L					
	M					

FOREIGN PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Country	Name	Classification[2]	
	N						
	O						
	P						
	Q						
	R						
	S						
	T						

NON-PATENT DOCUMENTS

		Include, as applicable: Author, Title, Date, Publisher, Edition or Volume, Pertinent Pages
	U	"Template Software Strengthens Core Product Family With Ease0of0Use and Functional Enhancements that Promote Unparalled Software Reuse", PR Newswire Assoc. 6/23/1997
	V J	Rational Software Corporation, "Using Rational Rose 4.0", Chapters 1 - 13, Released November 1996
	W	Rational Software Corporation, Rational Rose/C++ Round-Trip Engineering with Rational Rose/C++", pages 1 -227, Released November 1996
	X	"Busness Process Management WIth FLowMark", F. Leymann et al. IEEE, pages 230 - 233, 1994

* A copy of this reference is not being furnished with this Office action. See MPEP § 707.05(a). [1] Dates in MM-YYYY format are publication dates. [2] Classifications may be U.S. or foreign.

Notice of References Cited	Applicant/Patent: David Lars Ehnebuske et al.	Application/Control No.: 09/204,973	
	Examiner: Todd Ingberg	Art Unit: 2122	Page 3 of 4

U.S. PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Name	Classification[2]	
	A					
	B					
	C					
	D					
	E					
	F					
	G					
	H					
	I					
	J					
	K					
	L					
	M					

FOREIGN PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Country	Name	Classification[2]	
	N						
	O						
	P						
	Q						
	R						
	S						
	T						

NON-PATENT DOCUMENTS

		Include, as applicable: Author, Title, Date, Publisher, Edition or Volume, Pertinent Pages
	U	"IBM MQSeries WorkFlow", IBM Product description, NO DATE
	V	"Workfow-based applications", F.Leymann et al, IBM System Journal, Vol 36, No 1. pages 1 - 23, 1997
	W	"Business Language Analysis For Object-Oriented Information Systems", ,IBM Systems Journal, Vol 35, Issue 2, 1996
	X	"Business/ Enterprise Modeling", IBM Systems Journal, K.Robert, 1990

* A copy of this reference is not being furnished with this Office action. See MPEP § 707.05(a). [1] Dates in MM-YYYY format are publication dates. [2] Classifications may be U.S. or foreign.

Notice of References Cited	Applicant/Patent: David Lars Ehnebuske et al.	Application/Control No.: 09/204,973	
	Examiner: Todd Ingberg	Art Unit: 2122	Page 4 of 4

U.S. PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Name	Classification[2]	
	A					
	B					
	C					
	D					
	E					
	F					
	G					
	H					
	I					
	J					
	K					
	L					
	M					

FOREIGN PATENT DOCUMENTS

		Document Number Country Code-Number-Kind Code	Date MM-YYYY[1]	Country	Name	Classification[2]	
	N						
	O						
	P						
	Q						
	R						
	S						
	T						

NON-PATENT DOCUMENTS

		Include, as applicable: Author, Title, Date, Publisher, Edition or Volume, Pertinent Pages
	U	"The Impact of Object Orientation on Application Development", by A.A. Cockburn, IBM Systems Journal, v 32, No 3, 1993
	V	"Process Automation in Software Application Development", K.D. Saracelli et al., IBM Systems Journal, V 32, N 3, 1993
	W	"Object-Oriented Programming", R.P. Ten Dyke et al., IBM Systems Journal, v 28 N 3, 1998
	X	

* A copy of this reference is not being furnished with this Office action. See MPEP § 707.05(a). [1] Dates in MM-YYYY format are publication dates. [2] Classifications may be U.S. or foreign.

PRINCIPLES OF OBJECT-ORIENTED ANALYSIS AND DESIGN

JAMES MARTIN

P T R PRENTICE HALL
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data
MARTIN, JAMES (date)
Principles of object-oriented analysis and design / by James Martin.
p. cm.
"A James Martin book."
Includes bibliographical references and index.
ISBN 0-13-720871-5
1. Object-oriented programming (Computer science) I. Title.
QA76.64.M38 1993
005.1'1—dc20 92-31822
CIP

Editorial/production supervision: *Kathryn Gollin Marshak*
Liaison: *Mary P. Rottino*
Jacket design: *Lundgren Graphics*
Pre-press buyer: *Mary Elizabeth McCartney*
Manufacturing buyer: *Susan Brunke*

Published by P T R Prentice Hall
A Pearson Education Co.
Upper Saddle River, NJ 07458

ISBN 0-13-720871-5

Printed in the United States of America

10 9 8 7 6 5 4

ISBN 0-13-720871-5

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S.A., Mexico
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Pearson Education Asia Pte. Ltd., Singapore
Editoria Prentice-Hall do Brasil, Ltda., Rio De Janeiro

erence
White

9 EVENTS, TRIGGERS, AND OPERATIONS

EVENTS

To model the behavior of object-oriented systems, we determine what *events* happen. Events cause the systems to take various actions. We map the events and corresponding actions with an *event diagram*. The following are examples of events.

- The cat has had kittens
- Aunt Agatha has arrived for tea
- Traffic light turns to green
- An order is received
- An airline booking is requested
- Job is completed
- Invoice is transmitted
- Factory schedule is recomputed
- VCR tape rewind finishes
- CASE user created an object on the screen
- Nuclear forces went to DEFCON2 alert status
- Alarm clock goes off

The event diagram shows events and the operations triggered by events. End users tend to think intuitively about events and operations triggered by events. Instead, they should be taught how to read event diagrams and validate their correctness. The event diagram is a primary means of communicating OO behavior to end users.

A SEQUENCE OF OPERATIONS

An event diagram contains a sequence of operations. The operations are drawn with round-cornered boxes as in the following examples:

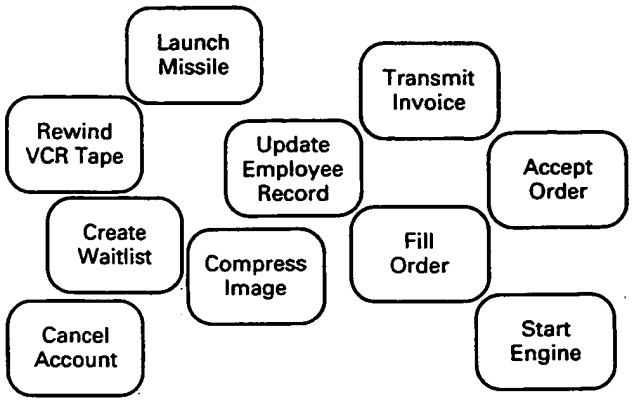


Figure 9.1 tells a story: a driver runs a red light and is chased by the police. In such a diagram, the sequence of operations is self-explanatory and can be easily understood by end users. We need only add some detail to show events, trigger rules, and control conditions.

The term operation refers to a unit of processing that can be requested. The corresponding procedure is implemented with a *method. The method is the specification of how the operation is carried out: it is the script for the operation.* At the program level, the method is the code that implements the operation.

Operations are *invoked.* An *invoked operation* is an *instance* of an operation. An operation may or may not change the state of an object. If it does, an *event* occurs.

EVENTS AND OPERATIONS

While some events are external to the system, most occur within the system and result from an operation. Events are drawn as small solid triangles. When an event results from an operation, the triangle is attached to the operation box, as shown on the next page.

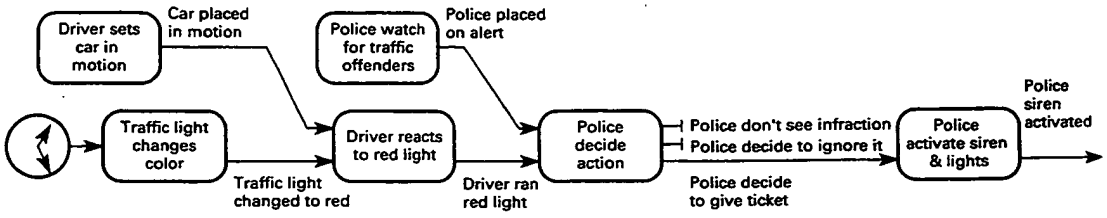
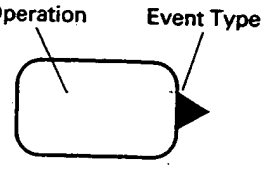
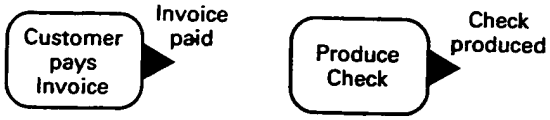


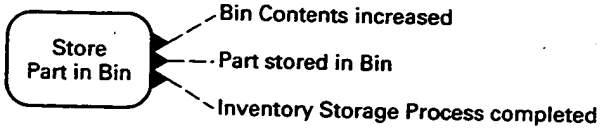
Figure 9.1 An event diagram for a car chase with all symbols and event subtype representation contracted.



The event occurs at a point in time. The small triangle represents this point in time when the corresponding state change occurs.



Sometimes, one operation results in multiple events:



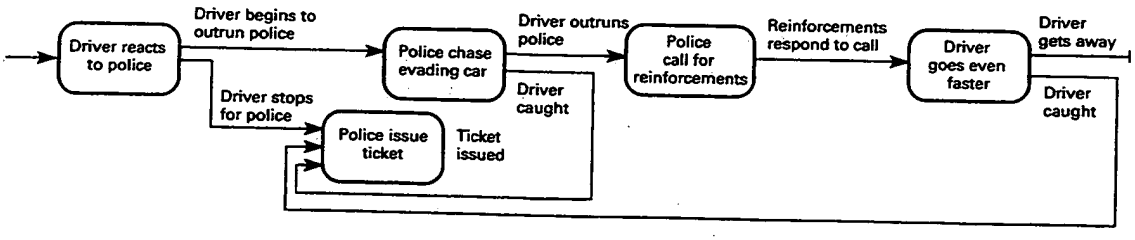
EVENTS AND STATE CHANGES

Events are, in effect, changes in the state of an object. Traffic Light turns to Green is a change in the state of the object Traffic Light. Job is completed is a change in the state of Job. An Order is received results in the creation of a new Order object. Invoice is transmitted is a change in the state of Invoice.

When the stock level of an item falls below its reorder point, this is a change in the state of a Stock object. Such an event also prompts us to act.

The process causing a state change may not be a part of the system we are building. However, the underlying object and its event may be important to the systems being specified.

> An event is a noteworthy change in the state of an object. Because of this, event diagrams must be tightly linked to state-transition diagrams.



When we create diagrams showing events, we think about the equivalent diagrams showing objects and states. Event diagrams are linked to state-transition diagrams. On the screen of a CASE tool, a change to an event diagram is linked to the corresponding state-transition diagram that itself relates to the object-structure diagrams.

THE LINKING OF OPERATIONS

A model showing the behavior of an object-oriented system has many operations linked together, as shown in Fig. 9.2.

> Operations result in events.
> Events cause other operations to occur.

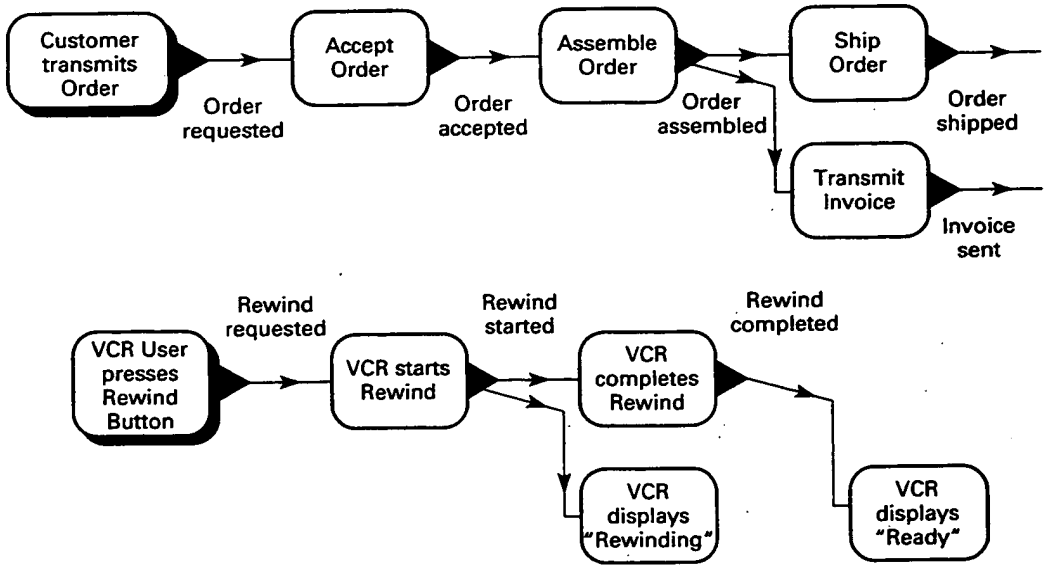


Figure 9.2

An operation may be performed by one *class*. The lines entering the operation often relate to the *requests* sent to that class. Sometimes, an operation is complex and needs expanding into more detailed event diagrams. At the lowest level of this expansion, *one* operation is usually performed by *one* class. One class has multiple methods that may be reflected by *multiple* operation blocks on event diagrams.

CAUSE-AND-EFFECT ISOLATION

Each operation carries out its task, regardless of what happens elsewhere. An operation is triggered by one

or more events, executes its method, and is expected to change the state of one object. The operation has no knowledge of what event triggered it and why. Additionally, it does not know what operations are triggered from its event(s). In short, it does not recognize its cause and effect—only that it is invoked to produce a state change of a given object. This isolation from cause-and-effect considerations is necessary for the operation to be reusable in many different applications.

> An operation has no knowledge of what triggered it and why.
> An operation does not know what operations are triggered by *its* result.
> The operation is isolated from cause-and-effect considerations.
> This isolation makes it reusable in different applications.

PRECONDITIONS AND POSTCONDITIONS

An operation has a precondition and postcondition. Preconditions and postconditions are of vital importance in helping to ensure that a system operates correctly.

> *Preconditions* are those conditions that must be true before an operation can take place.
> *Postconditions* are those conditions that must hold when the operation is completed.

If the precondition is not satisfied when an operation is invoked, then the operation should not execute; it returns an error message. The postcondition describes the result; if it is not correct after execution then the operation did not execute correctly (see Fig. 9.3).

Because an operation should have no knowledge of what triggered it or what will be triggered by its result, it should also have a precondition and post-

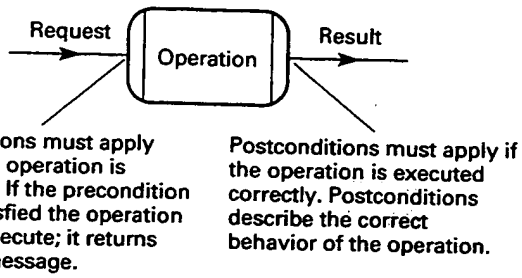


Figure 9.3 Preconditions and postconditions are independent of where the operation is executed.

condition that are independent of where it is used. The operation is implemented in software as a *method* in a *class*. The method may be invoked by different applications. It can provide a service to multiple *clients*. In effect it has a contract with its clients saying that if they send a request for which the precondition is satisfied, then the operation will execute so as to satisfy the postcondition.

> The presence of operation precondition and postcondition rules is, in effect, a contract that binds the operation. The operation says: "If you call me with the precondition satisfied, I promise to deliver a final state in which the postcondition is satisfied" [1].

A real-estate broker has a contract with its customers. The contract might have preconditions such as "I agree to pay the broker 6 percent of the purchase price if a property is purchased" and postconditions such as "A legal verification will have been made that the property has no encumbrances." This contract might be used by a broker for all of its clients, and the same contract might be used by many brokers.

In OO software design, a class, in effect, provides a contract to all potential users saying that if they invoke an operation with the correct precondition, it will provide a result satisfying the postcondition.

The existence of this contract simplifies programming. The program for the operation *is not required* to check the precondition. It can assume that the operation precondition rule is checked and is true before the operation executes.

Meyer comments:

> One of the main sources of complexity in programs is the constant need to check whether data passed to a processing element (routine) satisfy the requirements for correct processing. Where should these checks be performed: in the routine itself or in its callers? Unless module designers formally agree on a precise distribution of responsibilities, the checks end up not being done at all, a very unsafe situation or, out of concern for safety, being done several times.
>
> Redundant checking may seem harmless, but it is not. It hampers efficiency, of course; but even more important is the conceptual pollution that it brings to software systems. Complexity is probably the single most important enemy of software quality. The distribution of redundant checks all over a software system destroys the conceptual simplicity of the system, increases the risk for error, and hampers such qualities as extendibility, understandability, and maintainability.
>
> The recommended approach is to systematically use preconditions, and then allow module authors to assume, when writing the body of a routine, that the corresponding precondition is satisfied. The aim is to permit a simple style of programming, favoring readability, maintainability, and other associated qualities [1].

With an OO-CASE tool, it should be possible to point to an operation box and display its precondition and postcondition.

CLEAR MODULARIZATION

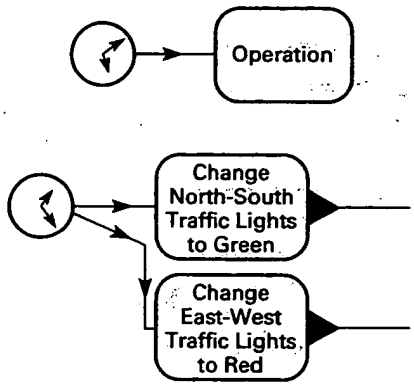
OO techniques provide two important ways to divide complex software into simple procedures. First, methods should result in a state change in one object. This state change, by itself, is usually simple and easy to program. Second, each operation is isolated from cause and effect. The operation can be reused on different software and on multiple interacting processors. The precondition and postcondition help to ensure its correct behavior.

OO techniques thus provide clear modularization that is simpler and more precise than that of conventional structured techniques. Objects of great complexity may be used with relatively simple requests, such as the use of a VCR. Complex objects can interact in one piece of software or on machines scattered across a network. Maintenance of software designed with OO techniques is easier than conventional software maintenance.

A world of difference exists between this clear modularization and the amorphous jelly-like nature of most non-OO software.

CLOCK EVENTS

A special type of event is a clock time being reached that triggers some operation. This type of event is drawn like a clock face:



EXTERNAL SOURCES OF EVENTS

Events are state changes that a system must know about and react to in some way. Often, many of the operations that cause these events are *external* to the system. In these circumstances, the operation symbol is drawn as a shadowed box with round corners as illustrated in Fig. 9.4(a).

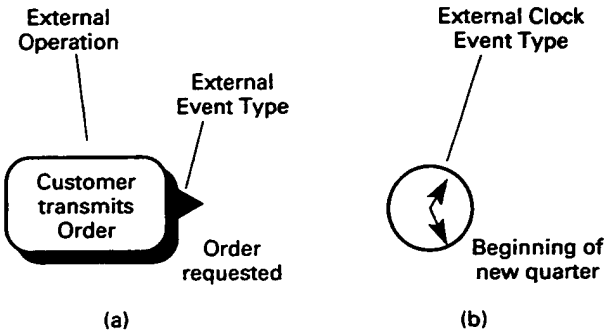


Figure 9.4 External operations are represented as round-cornered boxes with a shadow. When the event type is caused by an external clock, a clock face is used.

An *external clock* is a special form of external source. It indicates that an external process will emit clock ticks at some previously specified frequency, such as every second, the end of every day, the beginning of every month, or April 15th of every year. The event, then, is when the external clock tick occurs. External clocks are represented as clock faces.

Two external events resulting from an external operation can be seen in Fig. 9.5.

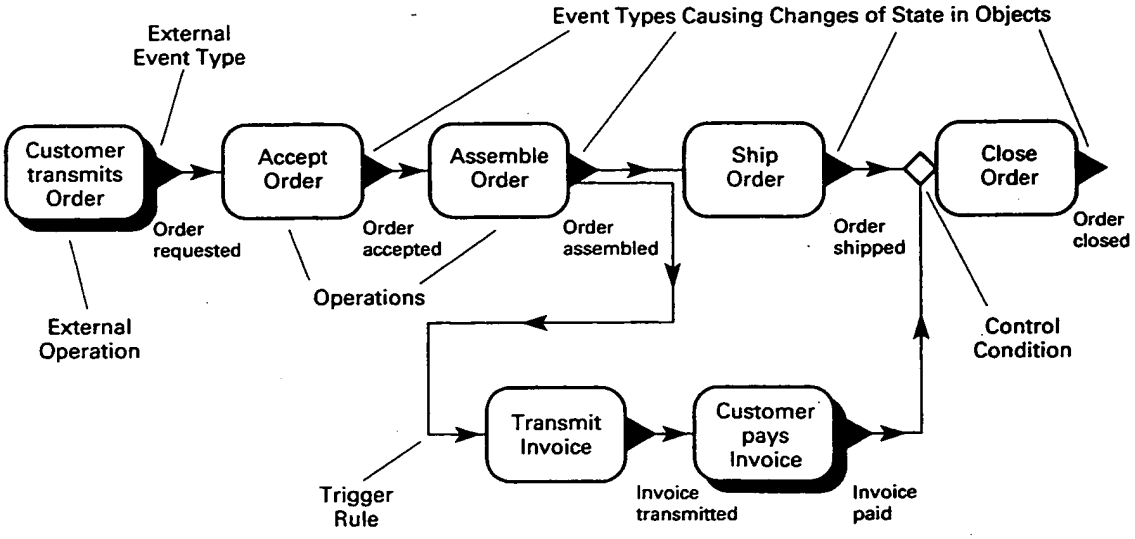


Figure 9.5 An event diagram showing operations and events that trigger operations. The events may relate to a corresponding state-transition diagram.

TYPES OF EVENTS The OO analyst does not wish to know about every event that occurs in an organization—only the *types* of events. Just as we talk about object types and instances of object types, so we

talk about event types and instances of event types. For example, the Waitlisted Booking Confirmed event type is the collection of those events in which an object changes from a Waitlisted Booking to a Confirmed Booking.

Event types indicate simple changes in object state, for example, when money is added to a bank account or an employee's salary is updated. Fundamentally, event types describe the following kinds of state changes:

- An object is *created.* For example, an airline booking is created.
- An object is *terminated.* For example, a product is destroyed or a contract is terminated.
- An object is *classified* as an instance of an object type. For example, a wife becomes a mother; a firm becomes a customer; an employee becomes a manager.
- An object is *declassified* as an instance of an object type. For example, a firm ceases to be a customer; a product is dropped from the sales catalog.
- An object *changes* classification. For example, a lawyer changes from associate to partner; an account changes from a normal account to an overdue account.
- An object's attribute is changed.

Events can associate one object with another. For example, in most organizations when an object is classified as an Employee, it must be associated with a Department. One event will classify the object as an Employee. A different event will create an association between the Employee object and a Department object. (Associations are objects like everything else. If named, this kind of association could be called the Employee-Department Assignment object type.) If the Employee changed Department, a new Employee-Department Assignment would be created and the old one terminated.

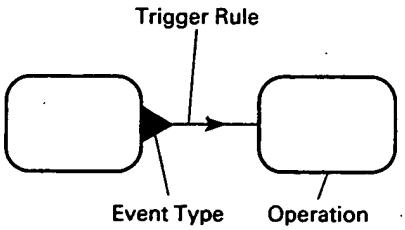
Updating an account balance is another example of an event. When $10 is added to account number 14274, an event associates the account with a different value.

Some events require other events to occur first. For example, before a Department can be closed, all of its Employees must be allocated to a different Department, the Offices it occupied must be allocated to a different use, and so on.

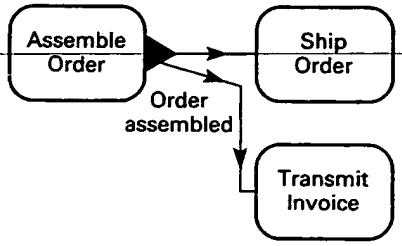
Sometimes one event causes a chain reaction of other events. Changing a node on the screen of a CASE tool, for example, may require a set of changes to other objects in order to preserve integrity. Adding a circuit to the wiring of a jumbo jet may require mandatory changes to many objects.

TRIGGER RULES

When an event occurs, it usually triggers an operation, as in the above diagrams; it *may* trigger multiple operations. The line going from the event to the operation it triggers represents a *trigger rule.*

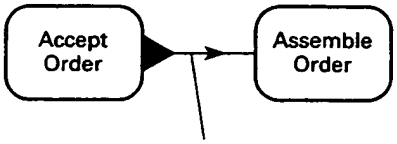


The *trigger rule* defines the causal relationship between event and operation. Whenever an event of a given type occurs, the trigger rule invokes a predefined operation. In the following diagram, the event type Order Assembled has two trigger rules—each of which triggers an operation:

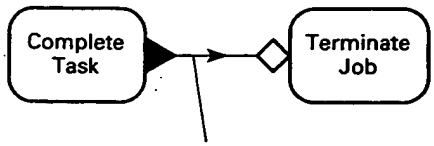


An event type may have many trigger rules—each invoking its own operation *in parallel.* Parallel operations can result simultaneously in multiple state changes.

A trigger rule invokes an operation and defines the necessary objects to be supplied to that operation.



Here, the Order object from Accept Order is passed as an argument to the Assemble Order operation.

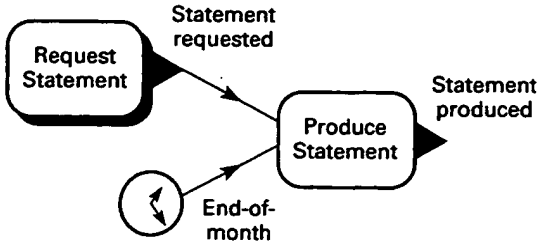


Given the Task object from Complete Task, its associated Job object must be determined by the trigger for invocation of the Terminate Job operation.

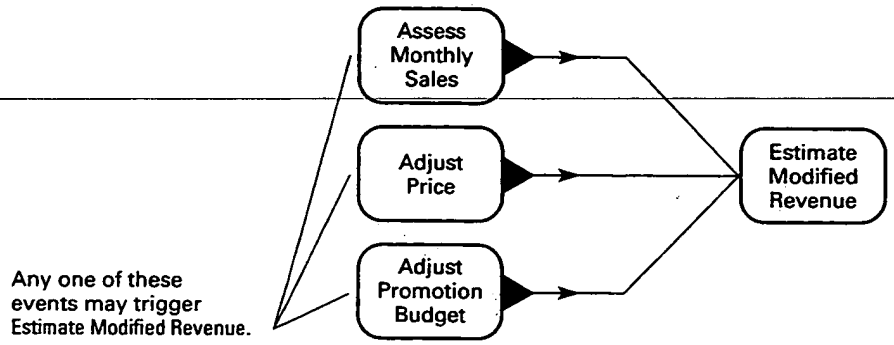
Trigger lines on an event diagram, then, indicate two things. First, they link the event to the operation that results from it. Second, they determine the objects required as arguments for the operation that the trigger invokes. These lines, therefore, define the rules for triggering an operation when a particular kind of event occurs. Because of that, they are called *trigger rules.*

MULTIPLE TRIGGERS

Sometimes an operation may be triggered in multiple ways. For example, Produce Statement may occur when a statement is requested or automatically at the end of the month:

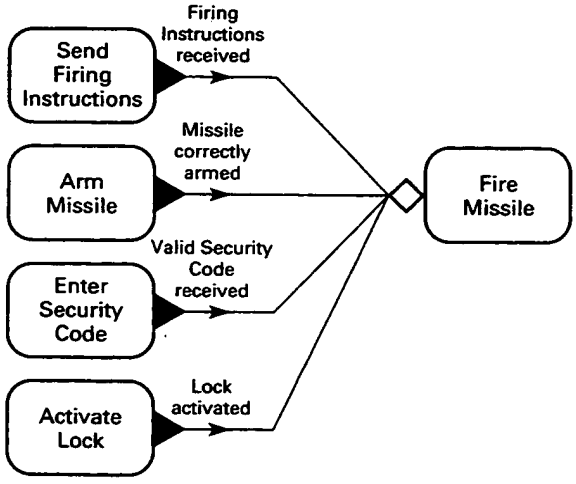


In the following illustration, any one of the three events can trigger Estimate Modified Revenue.

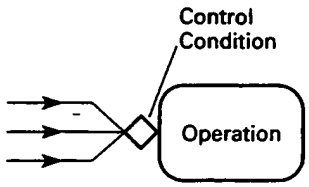


CONTROL CONDITIONS

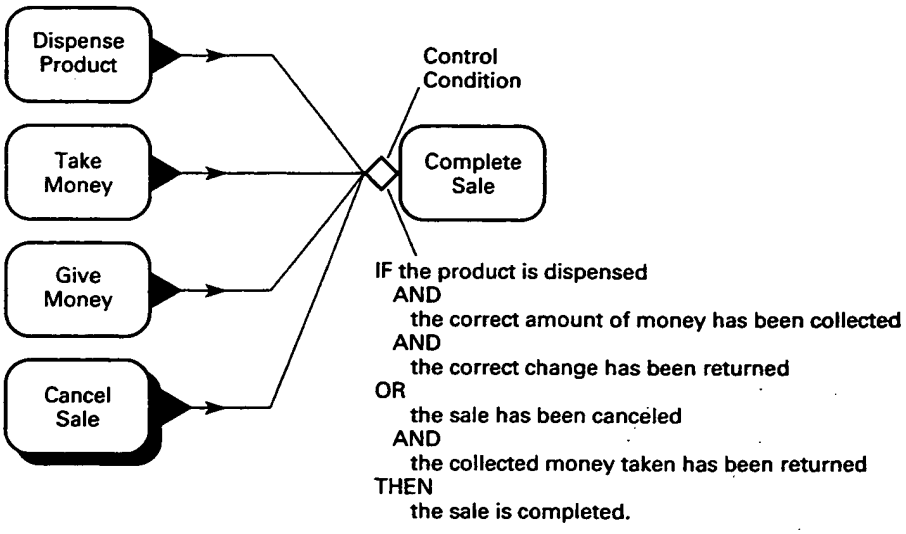
The above operations are invoked by one trigger rule (one event). Often, an operation requires multiple triggers to invoke it. For example, the operation Fire Missile can occur only if multiple conditions are obeyed:



The diamond in front of the Fire Missile block is referred to as a *control condition.*

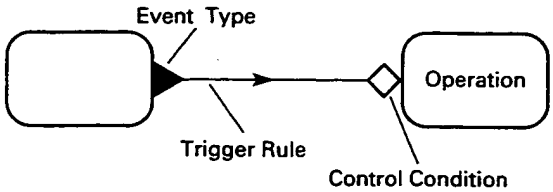


The control condition must be checked prior to invoking the operation. (The diamond shape is similar to the decision symbol used in flow charting.) The control condition can define a single condition or a complex collection of Boolean conditions. Control conditions are not necessarily just "and" conditions. They can involve elaborate conditions with "ands" and "ors":



THE BASIC CONSTRUCT OF

Event diagrams are thus constructed from operation blocks (see Fig. 9.6) and their connecting links as follows:



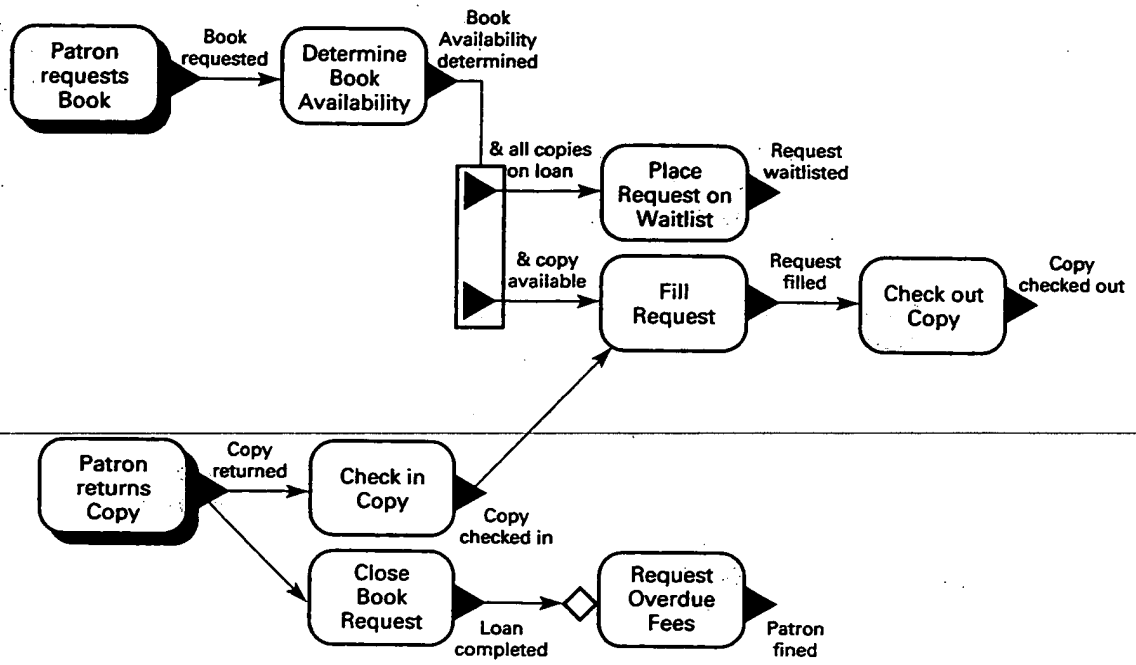
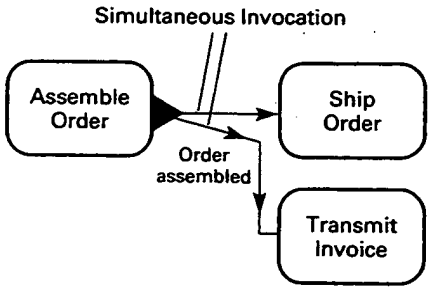


Figure 9.6 An event diagram for a library.

SIMULTANEOUS OPERATIONS

Event diagrams often indicate that multiple operations could be carried out at the same time. For example, in the following diagram Ship Order and Send Invoice could be done simultaneously.



Simultaneous operations might be done on separate computers on a network. Much future computing will employ parallel (concurrent) processing. Design techniques are needed to indicate what processing can be done simultaneously on separate processors. Event diagrams indicate when processing can occur in parallel.

Control conditions can also act as synchronization points for parallel processing. In other words, control conditions can ensure that a set of events is complete before proceeding with an operation. The control condition in Fig. 9.4, for instance, might state that Close Order cannot be done until the Order has been shipped and its Invoice paid.

In models of business processes there are many operations that occur concurrently. The model should show how they interrelate.

OPERATION SUBTYPES AND SUPERTYPES

We have emphasized that objects and classes are subtyped. The subtype inherits the properties of its parent which leads to substantial reusability.

In a similar way, operations are subtyped. The analyst should search for similarities in operations, so that the classes that perform the operations can inherit the same data and methods where possible.

Suppose that the operation Prepare Materials is followed by one of three operations. The operations are mutually exclusive, that is, only one of them can occur.

Mutual exclusivity can be represented by a branching line with a filled-in circle at the branch. As shown in Fig. 9.7, the circle looks like an "o" for "or" and means that the branches are mutually exclusive.

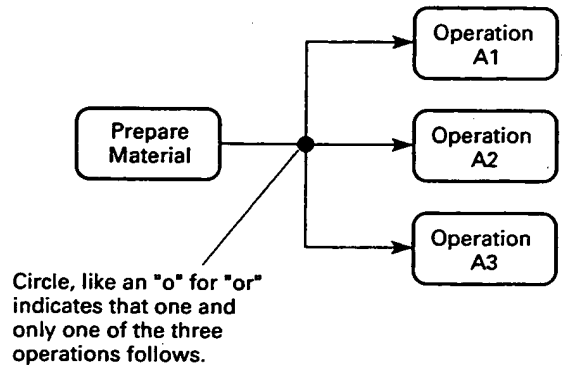


Figure 9.7

To represent this better, indicate that each of the three operations is a subtype of a general operation. In Fig. 9.8, they are shown in a partitioned box, as subtypes of Operation A.

Here, Operation 1, Operation 2, and Operation 3 all inherit properties for Operation A. This form of representation encourages the analyst to think about inheritance and reusability. It resembles the subtyping of objects illustrated in Fig. 7.4. This can lead to less redundancy—less code to design and generate. On larger scale examples, this can represent major savings and result in systems that can be changed more quickly.

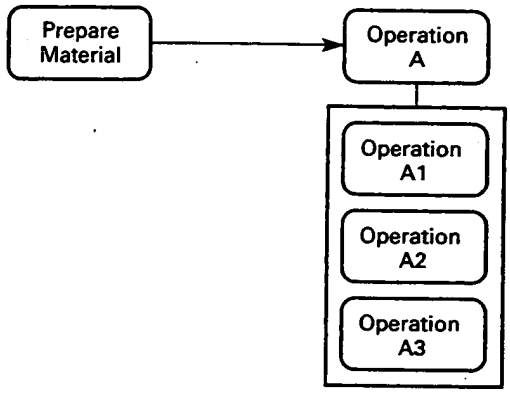
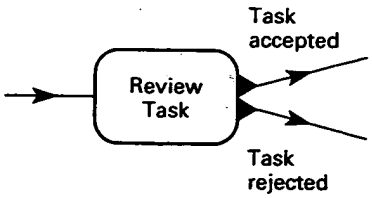


Figure 9.8

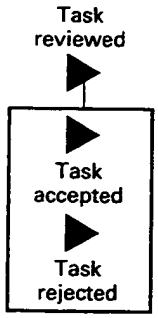
EVENT SUBTYPES AND SUPERTYPES

In a similar way, events are subtyped. This gives reusability of the trigger rules and control conditions.

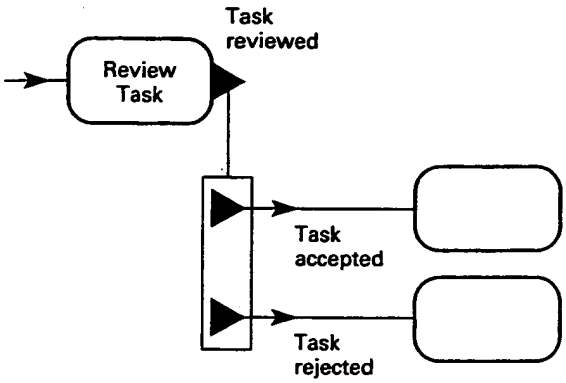
The following diagram shows two events resulting from the Review Task operation.



However, only one of the events can occur when a task is reviewed. The two event types are mutually exclusive. It is better, therefore, to show one event type Task reviewed with subtypes Task accepted and Task rejected.



The resulting event diagram is as follows:



As with the object-subtype boxes (see Chapter 6), the above event-subtype box indicates that the events in it are mutually exclusive. An event-subtype box (such as the object-subtype boxes in Fig. 6.8) can also indicate that other event subtypes might occur.

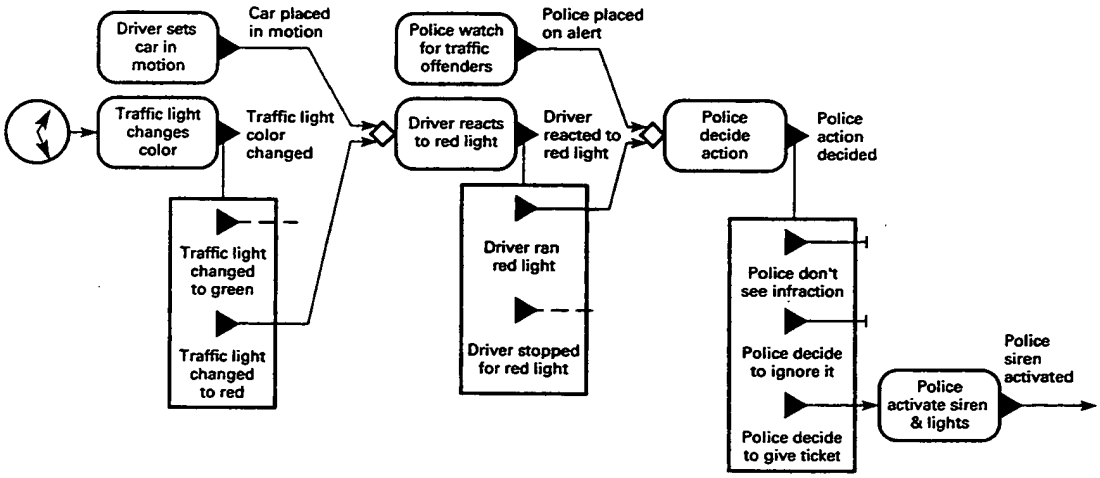
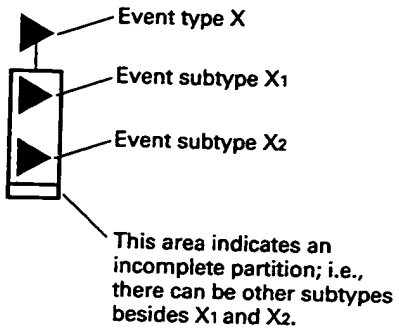


Figure 9.9 An event diagram for a car chase with event and control-condition symbols and with expanded event subtypes.



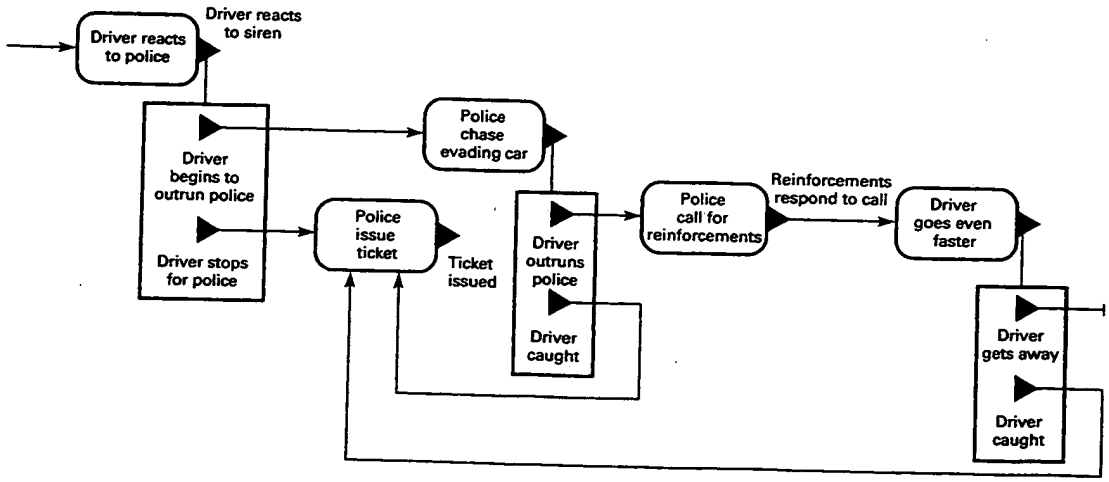
The car chase of Fig. 9.1 is redrawn in Fig. 9.9 showing events and event subtypes.

HIERARCHICAL SCHEMAS

The operation that causes an event to happen may be complex. The operation that rewinds a VCR tape appears simple—the VCR user just presses one button. When the Tape rewound event occurs, the user receives a simple message when the rewind is complete. However, internally, the rewind operation involves a whole set of operations and events, as illustrated in Fig. 9.10.

Figure 9.10 illustrates a hierarchical decomposition of event schemas. Boundaries can be placed around a complex event schema and treated as one high-level operation. The lower-level event schema, then, becomes the *method for the operation it decomposes.*

The events and operations that occur when the morning alarm clock goes off and a person prepares for the day are shown in Fig. 9.11.



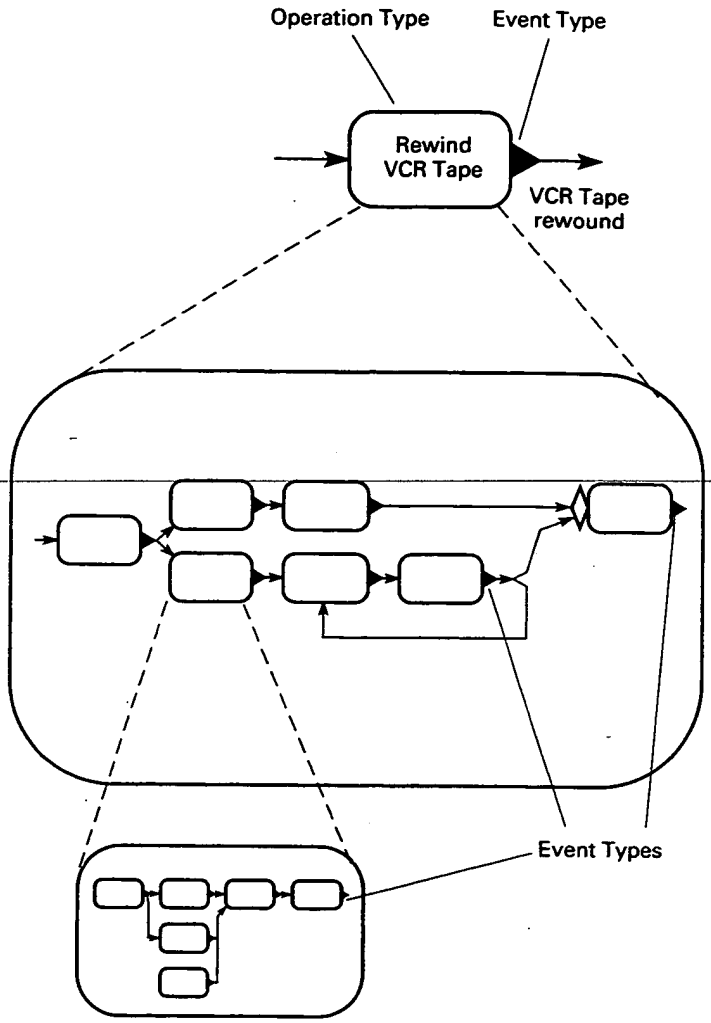


Figure 9.10 Rewinding a VCR tape is a simple operation for the VCR user. However, a complex set of operations and events have to occur inside the VCR in order for it to perform the rewind operation. They define the *method* for the rewind *operation*. Event diagrams may be decomposed hierarchically, like this, making a high-level operation appear simple.

OBJECT-FLOW DIAGRAMS

Event diagrams are appropriate for describing processes in terms of events, triggers, conditions, and operations. However, expressing large complicated processes in this way may not be appropriate. Often a system area is too vast or intricate to express the dynamics of events and triggers. Perhaps, in addition,

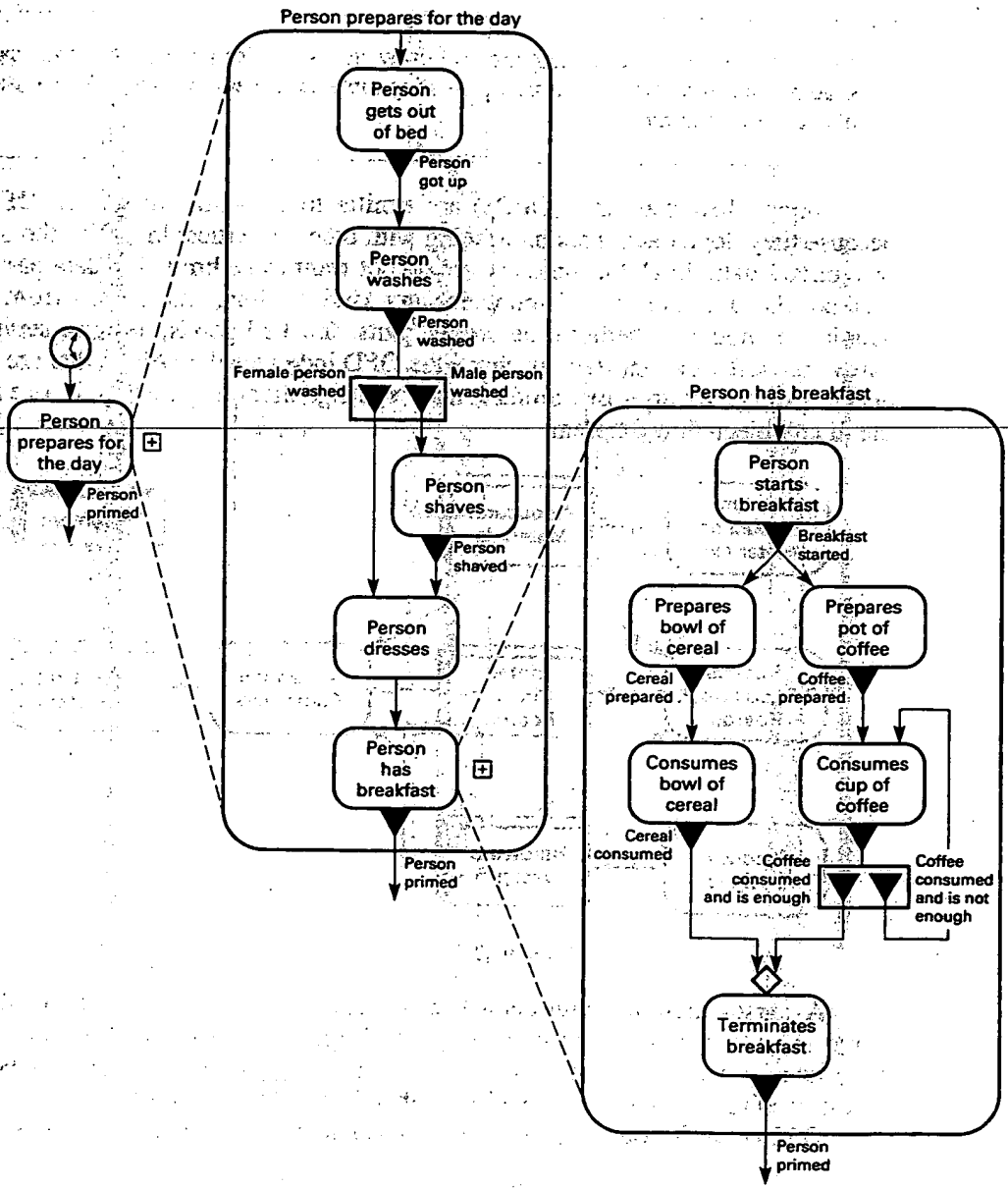


Figure 9.11 An operation for a person getting up in the morning. The window on the right shows the detail of the Person has Breakfast operation.

only a high level of understanding is necessary. This is particularly true of strategic-level planning. In situations such as these, an *object-flow diagram* is useful.

It is generally true that the diagrams we draw should be designed so that code can be generated from them. An *object-flow diagram* is an exception to that. It is a useful overview diagram.

Object-flow diagrams (OFDs) are similar to data-flow diagrams (DFDs), because they depict activities interfacing with other activities. In DFDs, the interface passes data. In OO techniques, we do not want to be limited to data passing. Instead, the diagram should represent any kind of thing that passes from one activity to another: whether it be orders, parts, finished goods, designs, services, hardware, software—or data. In short, the OFD indicates the *objects* that are produced and the activities that produce and exchange them. Figure 9.12 is an example of an object-flow diagram.

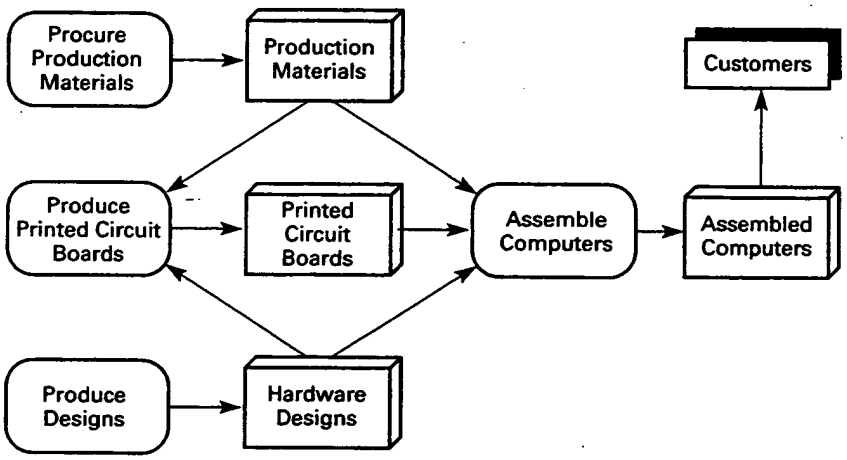
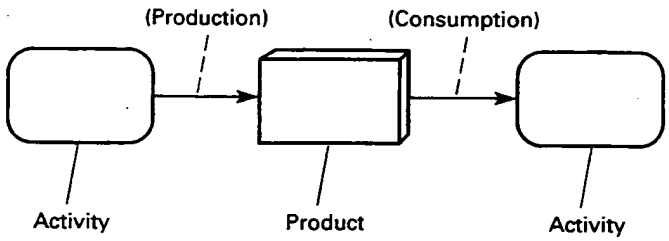


Figure 9.12 Object-flow diagram.

A person familiar with data-flow diagrams will recognize the round-cornered activity boxes, the shadowed, external-agent box, and the direction of the flow lines. Absent here, however, is the data-store symbol. A three-dimensional box is used to represent real-life objects that flow between activities.

OFDs describe objects and the way in which they are produced and consumed:



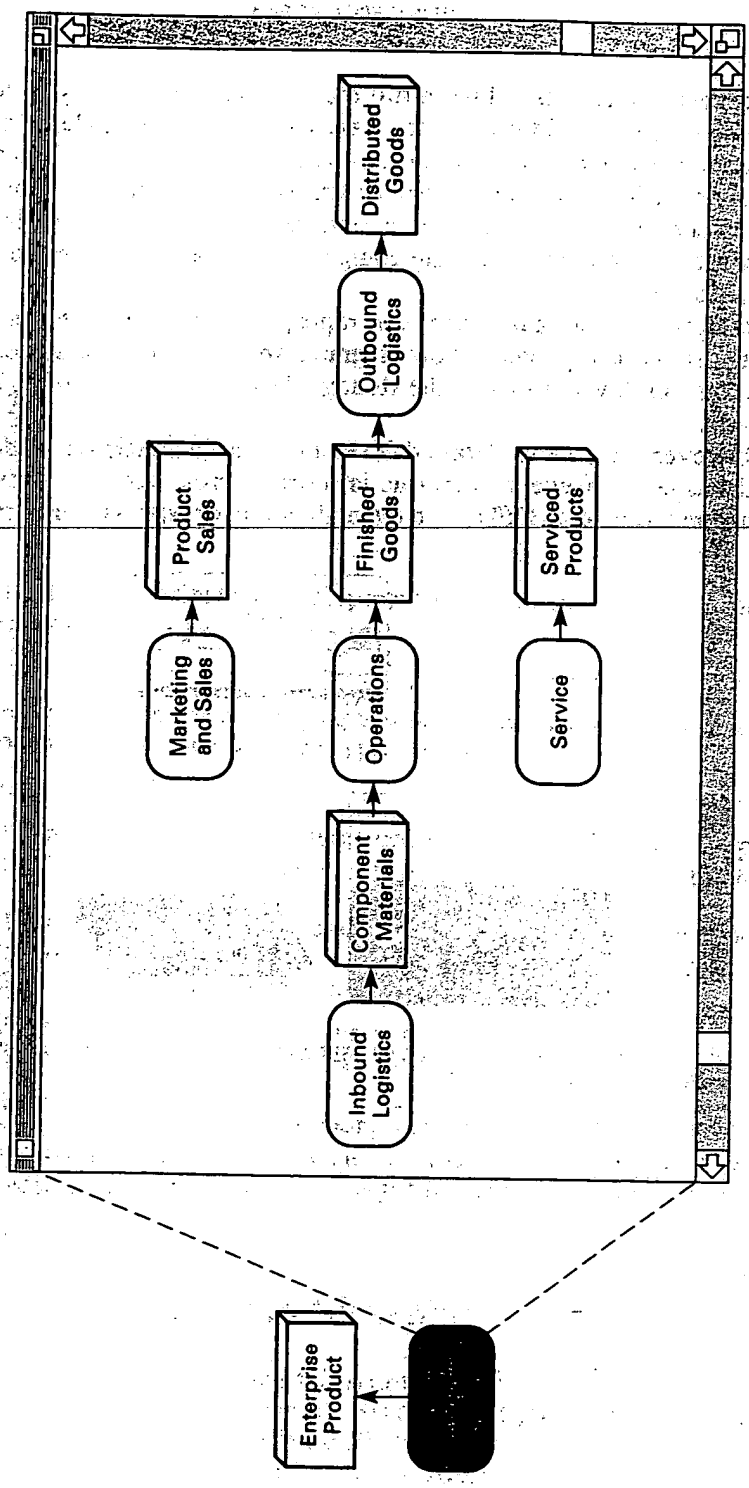


Figure 9.13 A decomposition of a manufacturing company's high-level primary activity.

The *product* is the end result that fulfills the purpose of the activity. Products move to other activities that add value to the product—to produce another more complex product. At every step, new qualities are created. In this way, the OFD can be used for strategic business planning as well as strategic information planning.

Any activity may be expanded into more detail. For example, the high-level primary activity of a manufacturing company could be represented in more detail as shown in Fig. 9.13.

Object-flow diagrams, then, can represent either top-down or bottom-up modeling. In particular, object-flow diagrams are very useful in modeling organizations in a top-down fashion at the strategic level. Activities can be decomposed into OFDs.

However, at a more detailed level in behavior analysis, expressing the dynamic aspects of event diagrams is more appropriate. An activity can be expressed in terms of an OFD or an event diagram, or both as shown in Fig. 9.14.

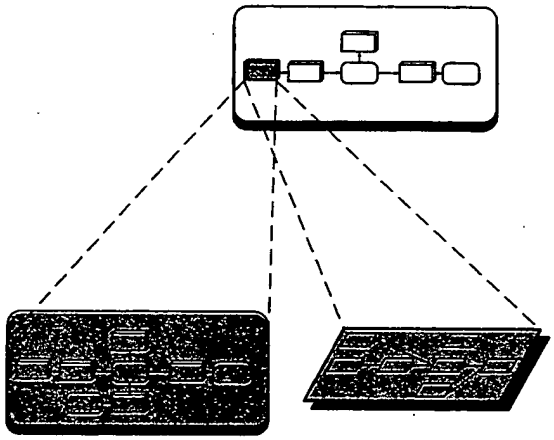


Figure 9.14 Each activity can be expressed as an object-flow diagram, an event schema, or both.

The event diagram expresses a process in a more rigorous fashion that can generate code. To represent basic control structures and processing flow, and when the dynamics of events and triggers are not yet comprehensible, the object-flow diagram is useful.

REFERENCE

1. Meyer, Bertrand, *Object-Oriented Software Construction,* Prentice Hall, Englewood Cliffs, NJ, 1988.

10 RULES

One goal of object-oriented development ought to be to avoid programming wherever possible.

> Wherever possible, the code for systems should be generated from models that are easy for end users to understand and experiment with.

The desired behavior of systems can be described with the help of *rules*. Business policies, for example, can be expressed in rules such as the following:

WHEN a customer has bought more than twice the average sales per customer
for the previous 12 months
THEN it is categorized as a good customer

WHEN stock falls below reorder point for a product
THEN only good customers have their orders processed immediately

WHEN new stock arrives
IF orders are on hold
THEN orders are sorted by customer rating then earliest order date, and filled
in that sequence

WHEN orders are filled
IF the customer is a bad payer
THEN the order is put on hold until the amount due is received
AND the customer is notified

RULES EXPRESSED IN ENGLISH

Rules need to be rigorous so that they form a basis for code generation. However, it is particularly important that rules are understandable by end users. End users

must be able to check that the rules correctly represent business policies and desired system behavior. Rules should therefore be expressed in English (or the national language of the end users).

A typical rule expression in the language Prolog is as follows:

```
sister (x,y) : - female (x), parent (x,z), parent (y,z)
```

This is not likely to be understood by most end users. (It means x is the sister of y if x is female and if x has a parent, z, and y has the same parent, z.)

When we use rules in object-oriented modeling, the rules can be expressed in English but at the same time be rigorous. English-language rules can be built with a rule editor that is part of an OO-CASE toolset. At the same time that the English is created, code is generated so that the rule can be executed immediately. The resulting English may be clumsy and tedious for end users to comprehend. When this is so, the analyst should write a more elegantly phrased sentence expressing the same rule. We then have both formal and easy-to-read English in windows which are linked to an OO model that end users can understand and work with. Their business policies are expressed in such a model, and code is generated directly from the business policies. This is done, for example, with the OO-CASE tool OMW, Object Management Workbench, from Intellicorp.

DECLARATIVE VERSUS PROCEDURAL STATEMENTS

We can distinguish between *declarative* and *procedural* languages or statements.

Conventional programming languages are *procedural*. They give a set of instructions that a computer must execute in a specified sequence. The sequence may vary depending on conditions tested. Groups of instructions can be executed repetitively (loops).

Declarative languages declare a set of facts and rules. They do not specify the sequence of steps (procedure) for doing processing. The computer uses the facts to derive a program for a particular procedure. The facts may be expressed in various ways. They may, for example, be in the form of a record:

Book	Author	Publisher
Future Shock	Toffler	Random House

Facts may be values that populate a spreadsheet. They may be expressed with statements such as the following.

Pathogens associated with Gastrointestinal Tract include

- Enterococcus
- Clostridium Gangrene

- Bacteroids
- Klebsiella
- Pseudomonas
- E.Coli
- Enterobacterium
- Proteus

Facts may also be expressed with equations, for example:

```
PRINCIPAL = INSTALLMENT X 100/INTEREST_RATE (1 –
(1 + INTEREST_RATE/100) * * – N) (1 + INTEREST_RATE/100)
```

If *any* three of the four variables in this financial equation are entered, the fourth will be calculated. *The user does not give the sequence of steps in doing the calculation.* Similarly, several simultaneous equations could be used.

End users, who generally have difficulty understanding programs and checking their correctness, can easily understand statements of facts and rules.

> Declarative statements are much easier to grasp and validate than procedural language.
>
> Where possible, we should build systems from declarative statements linked to OO diagrams that end users can understand.

To a large extent, but not completely, the design of an OO system can be created automatically from facts and rules of the type described in this chapter, and code can be automatically generated.

MAKING BUSINESS KNOWLEDGE EXPLICIT

The rules described in this chapter capture the know-how about how the business or system should operate.

> Rules are encapsulated business knowledge.

We need techniques with which business know-how can be captured, made explicit, made easy to read, and translated directly into executable code. With traditional structured techniques, business policies are not made explicit. They become buried in code written in COBOL or other languages. One rule is often reflected in multiple programs and may be virtually impossible to extract from

those programs by examining the code. When the policy changes, it is difficult to change the code or even to know what code ought to be changed.

> We need to make business policies explicit and translate them directly into code. When the policies change, it should be possible to regenerate the code quickly.

Our challenge is to find diagrams that are meaningful to executives and business people, that enable such people to visualize how their enterprise operates and to help redesign it where necessary. An OO-CASE tool should enable its users to navigate through the diagrams and expand windows that show the rules.

> Collectively, the OO diagrams and rules should represent the laws about how the business is run.

With traditional structured techniques, there is a poor translation of business policies into programs. With OO techniques and rules, we want the most direct translation of business policies into generated code. When business policies are changed, we want that to be reflected quickly in regenerated code.

RULES LINKED TO DIAGRAMS

We have emphasized the value of OO-CASE tools for OO modeling, analysis, and design. CASE tools use the diagrams described in Chapters 6 to 13. Rules should be associated with these diagrams. The CASE tools may show them in windows linked to the blocks or symbols on the diagrams.

Some rules are associated with the diagrams used in Object Structure Analysis (described in Chapters 6 and 7):

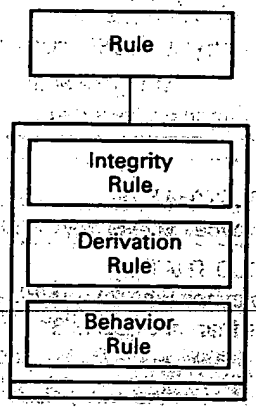
- Inheritance and subtyping diagrams
- Object-relationship diagrams
- Composed-of diagrams
- Diagrams showing data structures

Other rules are associated with the diagrams used in Object Behavior Analysis (described in Chapters 8, 9, and 13):

- Event diagrams
- State-transition diagrams
- Diagrams showing methods

CATEGORIES OF RULES

Several different types of rules exist. Business rules can be classified as integrity rules, derivation rules, and behavior rules [2].



Integrity rules state that something must be true. For example, a value must be within a certain range, an object relationship must have a stated cardinality, a precondition must hold before an operation is executed, and so on. *Operation precondition rules* and *operation postcondition rules* are important special cases of integrity rules.

Integrity rules should start with the phrase "it should always hold that." Box 10.1 gives some examples of integrity rules.

Derivation rules state how a value or a set of values is computed. Box 10.2 gives examples of derivation rules.

Behavior rules describe dynamic aspects of behavior. They are commonly associated with event diagrams. They often express business rules that describe what conditions must be true for an operation to be triggered.

In some methodologies, cardinality constraints are referred to as "business rules" and are the only form of business rule. These should be referred to as "cardinality rules" to indicate that they are only one narrow type of business rule.

STIMULUS/RESPONSE RULES

Stimulus/response rules express triggering behavior and generally have the form

```
WHEN    <event>
IF      <condition to fulfill>
THEN    <operation>
```

In a business model this can be

BOX 10.1 Examples of integrity rules.

These are easy-to-read versions of rules, which are in more formal English when created by a CASE rule-builder. The same applies to Box 10.2.

IT MUST ALWAYS HOLD THAT
The number of employees is less than 1,000

IT MUST ALWAYS HOLD THAT
The number of employees who are managers and earning a salary greater than 100,000 is less than or equal to 3

IT MUST ALWAYS HOLD THAT
The number of hotels (with the same hotel name and located in the same resort) is less than or equal to 1

IT MUST ALWAYS HOLD THAT
A manager is not a secretary

IT MUST ALWAYS HOLD THAT
IF an employee has a company car
THEN this employee does not receive a monthly transport reimbursement

IT MUST ALWAYS HOLD THAT
IF an employee works on a project
THEN this employee works for a department

IT MUST ALWAYS HOLD THAT
IF an employee has a marked parking slot
THEN this employee has a company car and is a manager with a salary greater than 60,000

IT MUST ALWAYS HOLD THAT
IF a flight is scheduled to depart from City C
THEN this flight is not scheduled to arrive in City C

AFTER modify of salary of an Employee.E IT MUST HOLD THAT
The new salary of Employee E is greater than the old salary of an Employee E

IT MUST ALWAYS HOLD THAT
The age of an employee is less than 65

IT MUST ALWAYS HOLD THAT
The sum of salaries of employees working for Department D is less than 0.6 * budget of Department D

BOX 10.2 Examples of derivation rules.

PRODUCT.NET_PRICE = PRODUCT.PRICE * (1 + TAX_PERCENTAGE/100)

DERIVE Employee reporting to Manager M
as Employee working for Department managed by Manager M.

Invoice.Total = Sum of Invoice.Line_totals + Sale_tax + Service_charge

---

WHEN Invoice.Net_total > 0 and < 2500
Taxes = Invoice.Net_total * 0.15
ELSEWHEN Invoice.Net_total >2500 and < 50000
Taxes = (Invoice.Net_total - 2500) * 0.18 - 3750
ELSEWHEN Invoice.Net_total.> 50000 and < 75000
Taxes= (Invoice.Net_total - 50000) * 0.30 - 8250
ELSEWHEN Invoice.Net_total > 75000 and < 100000
Taxes= (Invoice.Net_total - 75000) * 0.40 - 15750
ELSEWHEN Invoice.Net_total.> 100000
Taxes= (Invoice.Net_total - 100000) * 0.46 - 25750

IF Symptom_of_Problem is equal to "biscuits pale" or
Symptom_of_Problem is equal to "biscuits not risen"
THEN set Hypothesis_of_Problem to "low temperature gases"

IF a customer has bought more than twice average sales per customer for the last 12 months
THEN classify customer as a "Good Customer"

WHEN immediate shipment cannot be accomplished for all batches
THEN give priority to batches containing items for a Good Customer

---

WHEN immediate shipment cannot be accomplished for all batches containing items for a Good Customer
THEN give priority to batches with nearby delivery requirements

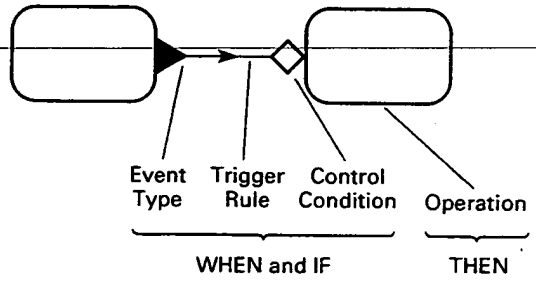
WHEN	an event happens
IF	a condition is fulfilled
THEN	do something

In a technical system, it can be

ON	<stimulus>
IF	<condition>
THEN	<response>

In these expressions the IF clause is optional.

This form of expression relates to the basic construct of the event diagram:



Such stimulus/response rules can be expressed rigorously but can also be translated into an English statement that end users can validate. For example

```
WHEN Account is posted to Sales Ledger
IF Customer is of type D
   and Balance has been negative for 18 days
THEN set Status to Credit Stopped.
```

The THEN part of a rule is an operation. For example, set Status to Credit Stopped is an operation. Often, however, the THEN part of a rule invokes an operation by name. For example, when a stock item falls below the reorder point, send a request to Reorder the stock item.

OPERATION CONDITION RULES

As commented in the previous chapter, each operation has its own context-independent precondition and postcondition. Preconditions and postconditions should be expressed with rules that end users can understand and verify. These rules are of vital importance in helping to ensure that software operates correctly. Bertrand Meyer states that the presence of these rules in a routine should be

viewed as a contract that binds the routine and its callers [1]. It is, in effect, a contract relating to software reliability.

Operation precondition rules express those constraints under which an operation will perform correctly. The operation cannot go ahead unless these constraints are satisfied. Operation precondition rules have the form

Order a Product
ONLY IF there is an authorized Supplier offering this Product

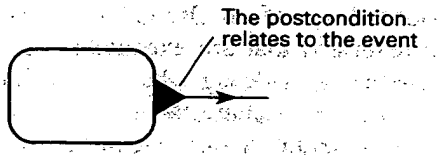
Promote Staff Employee to Manager
ONLY IF Employee has a Staff position
and Employee is not a Manager

In contrast, the *operation postcondition rule*, in effect, guarantees the results. It says that when the operation is executed, certain state changes will occur. Operation postcondition rules have the form

Order Product from supplier IS CORRECTLY COMPLETED
ONLY IF the Product Order for supplier is created

Promote Staff Employee to Manager IS CORRECTLY COMPLETED
ONLY IF Employee is not in a Staff position
and Employee is a Manager

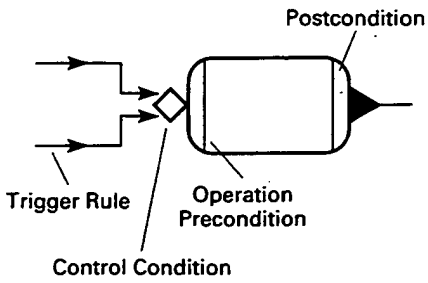
Events are those state changes to which a system must react. Therefore, *events* (represented by a black triangle), reflect an aspect of an operation postcondition rule. When that aspect is true, the event occurs:



CONDITIONS FOR EXECUTING AN OPERATION

Before an operation can be executed, two things must be true: its precondition, which is *independent* of the particular application, and its control condition, which is *dependent* of the application. Both of these are expressed as rules.

The user of an OO-CASE tool should be able to point at the control condition diamond or at the operation box and display the control condition rules or the operation precondition rules.



The control condition relates to the trigger rules and is often designed to achieve correctness of the precondition.

These rules, displayed in windows on the event diagram, should reflect how the business people want to run the business. Business policies are explicitly stated.

EXECUTABLE DIAGRAMS

The event diagram, including statements of rules to express preconditions, triggers, control conditions, and the calculations or transformations done in an operation, is an executable diagram, that is, it can be converted to program code. The operation itself may be expressed declaratively or in a form that can drive a code generator.

> An event diagram with rules is executable.
>
> The diagrams of traditional structured analysis are not executable.

The most useful form of an OO-CASE tool is one that generates code immediately from diagrams that are executable. The code can be immediately executed to see whether it is doing what the designer intended—rather like a spreadsheet tool. When you create the columns, rows, and calculations of a spreadsheet, you can immediately run it, then quickly change it, and rerun it. You can experiment with it, adjusting its design.

An OO model is likely to be much more complex than a spreadsheet but the tool should make it equally interactive. You should be able to build it and immediately run it, add instances of class data, and observe its behavior, grow it, and change it quickly. You should be able to experiment with your design as you build it.

At first, the code will be generated interpretively and need not be machine-efficient. Later, when the design works well, it can be compiled and perhaps redesigned for maximum efficiency.

The ability to run the design immediately, experiment with it, and change it, greatly increases creativity. It expands our ability to invent interesting software.

RULES ATTACHED TO OTHER OO DIAGRAMS

Rules can be linked to other types of diagrams as well as event diagrams.

> Rules can be attached to any of the diagrams in the previous chapters.

Examples of rules associated with the diagrams of OO analysis are shown in Box 10.3.

Rules can be divided into two types—object-state rules and object-behavior rules. Object-state rules are identified in object-structure analysis. They are associated with diagrams, such as the data-structure diagram, object-relationship diagram, or composed-of diagram. Object-behavior rules are identified in object-behavior analysis. Most are associated with the event diagram; some are associated with the state-transition diagram.

Figure 10.1 shows a variety of OO diagrams with rules linked to them, as they should be in an OO-CASE tool. The rules may be expressed both in formal notation and easy-to-read English, so that either may be displayed.

Figure 10.2 shows a rule window linked to an event diagram in the IntelliCorp Object Management Workbench (OMW). A variety of such rule windows can be connected to the diagrams this tool uses. With a rule editor the analyst builds rules in English and at the same time generates executable code for the rules. The analyst can immediately run his model, animate it, and generate spreadsheets of values from it. He can thus immediately test and modify the model. The business people using the tool, possibly in facilitated workshops (discussed later), can try out different business policies and observe their effect. They can thus redesign business processes and observe the effects of the redesign.

THE PROGRAMMER AS LAWYER

The English expression of rules precise enough for code generation is formal and sometimes difficult for business people to read. The analyst should therefore translate the executable rules into easy-to-read English.

The bottom box of the Rule Editor Probe window in Fig. 10.2 contains the formal English. The box above, labeled "Meaning," contains the analyst's easy-to-read version of the same rule. The "Comment" box above this may contain comments about who wants the business policy that the rule represents, or why it is used.

Figure 10.3 shows three more rule windows that are linked to appropriate items in OO diagrams. The red item in each diagram is a rule in formal English that is created with the tool's rule-builder. As the Rule Editor assists the analyst in building the rule it simultaneously generates executable code. The analyst can compose an easy-to-read version of the rule in the box labeled "Meaning."

BOX 10.3 Examples of business rules associated with the diagrams for OO analysis.

Object Structure Analysis

- *Rules associated with attributes,* such as
  - — A Customer is good if it has bought more than twice the average sales per customer for the last 12 months.
  - — When Employee has Salary > 150,000 it can have stock options.
- *Rules associated with object-relationship diagrams,* such as
  - — A good Customer can place any number of Orders.
  - — A bad Customer can place no more than 10 Orders.
- *Rules associated with composed-of diagrams,* such as
  - — A Lighting Track has a maximum of 12 spotlights.

Object Behavior Analysis

- *Rules associated with transitions,* such as
  - — When Salary is updated new value must exceed old value
- *Rules associated with event diagrams*
  - — Trigger rules, such as
    When Stock < Reorder Level then reorder
  - — Control condition, such as
    If Security Code is correct then . . .
    If time is between 9 AM and 5 PM . . .
    If good Customer . . .
- *Complex rules expressed with event diagrams,* such as
  - — When Immediate Shipment cannot be accomplished for all Batches, then give priority to Batches containing items for Good Customers.
  - — When Immediate Shipment cannot be accomplished for all Batches containing Items for Good Customers, then give priority to Batches with nearby Delivery Requirements.

From the model and its attached rules a spreadsheet is created automatically. The analyst can populate the spreadsheet with values and observe the effects of the rules. The operations on the event diagram may be resequenced, the business policies changed, and the spreadsheet changed accordingly. Graphics and charts can be generated from the spreadsheet. Business people can thus explore

the behavior of the model, the effects of their policies, and the possibilities for business process redesign.

With this approach, business people can express their business policies in normal English. The analyst, rather like a lawyer, translates this into precise English and corresponding code. When the business people change a policy, this change can be translated directly and quickly into software that implements the new policy.

THE INFERENCE ENGINE

In the 1980s, artificial intelligence systems became highly fashionable. The primary mechanism of such systems built in the 1980s was an *inference engine.*

An inference engine uses a collection of facts and rules about a specific area of knowledge and makes deductions using the techniques of logical inference. It can respond to a request by selecting rules and firing them, effectively chaining the rules together to perform inferential reasoning. It may use forward chaining (input-directed reasoning), backward chaining (goal-directed reasoning), or both. It enables a computer to make complex deductions without programmers having to code an application.

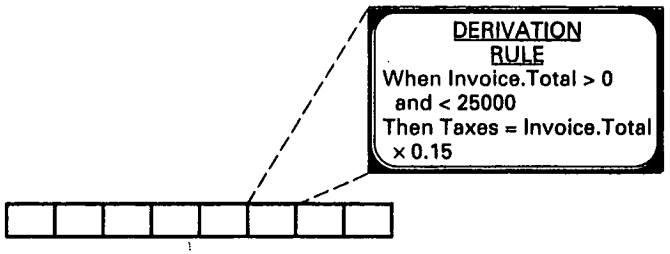
The concept of the inference engine was used to build *expert systems.* Expert systems store knowledge culled for an expert in the form of facts and rules and make deductions by employing an inference engine. The goal is to make the computer give advice like a human expert in a specific narrow domain of knowledge.

The rules that an inference engine chains together to make deductions are called *production rules.* The rules referred to in this chapter are generally not production rules. They are rules that are linked to OO diagrams in order to generate meaningful models and code.

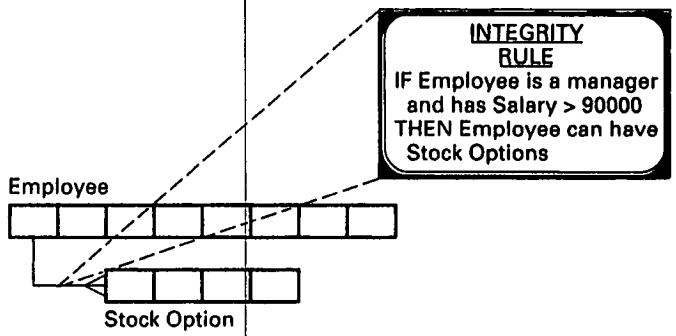
An inference engine, like any other computing technique, may be used to implement *methods* in a *class.* Usually, this requires a *small* collection of rules rather than the large rule collections that characterize some artificial-intelligence systems. Each method makes a relatively simple change to the data in the class. Because it uses a small collection of rules (usually fewer than 100), it can be relatively fast and efficient.

Some expert systems were built with a large number of rules. There was an ill-thought-out notion that if an inference engine could use a vast collection of rules, it could solve formidable problems. Some early expert systems used more than 10,000 rules. Large amorphous collections of rules present two problems. First, machine performance is bad. Inferencing algorithms do not scale up efficiently when there are thousands of rules. Second, it is difficult to tell what the inferencing system is doing and hence it is difficult to debug it. Changing the behavior of a rule-based system is very easy, in principle. One simply changes the rules—which is much easier than changing a COBOL program. However, its behavior is difficult to understand if the collections of rules are large and unstructured.

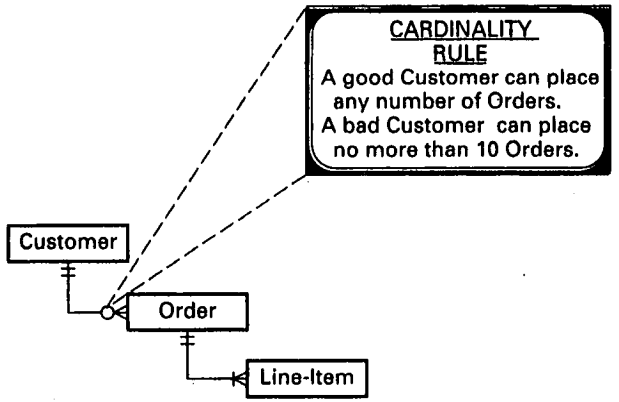
RECORD STRUCTURE



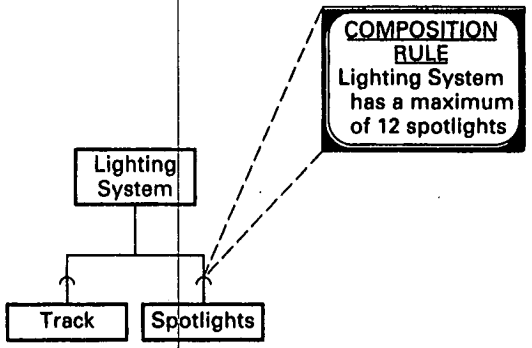
DATA STRUCTURE DIAGRAM



OBJECT-RELATIONSHIP DIAGRAM



COMPOSED-OF DIAGRAM



OPERATION

EVENT DIAGRAM

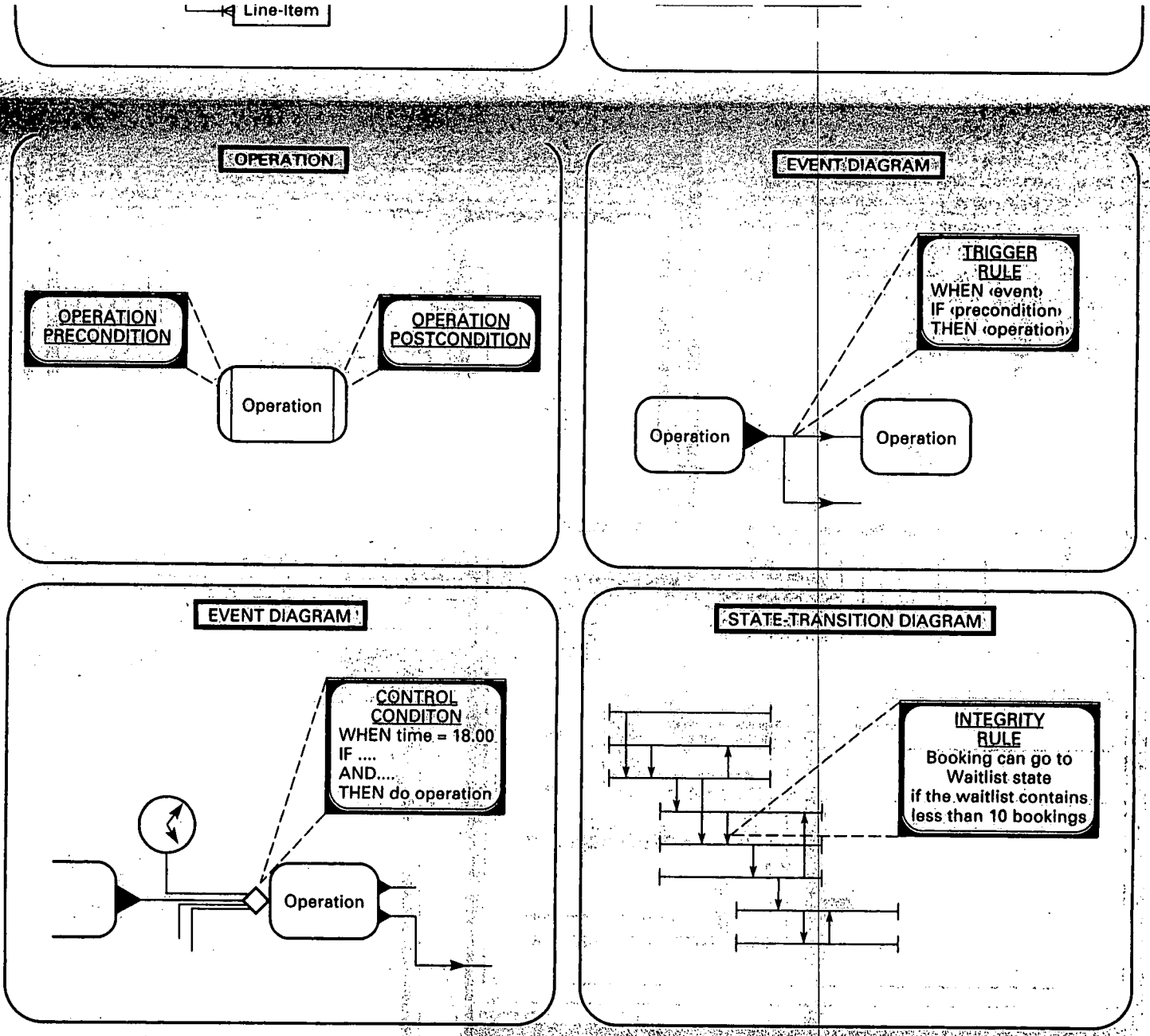


Figure 10.1 Rules linked to OO-CASE diagrams. The formal English which a CASE rule-builder creates is less easy to read than some of these illustrations. The analyst supplements the formal English with easy-to-read English to aid communication with business people.

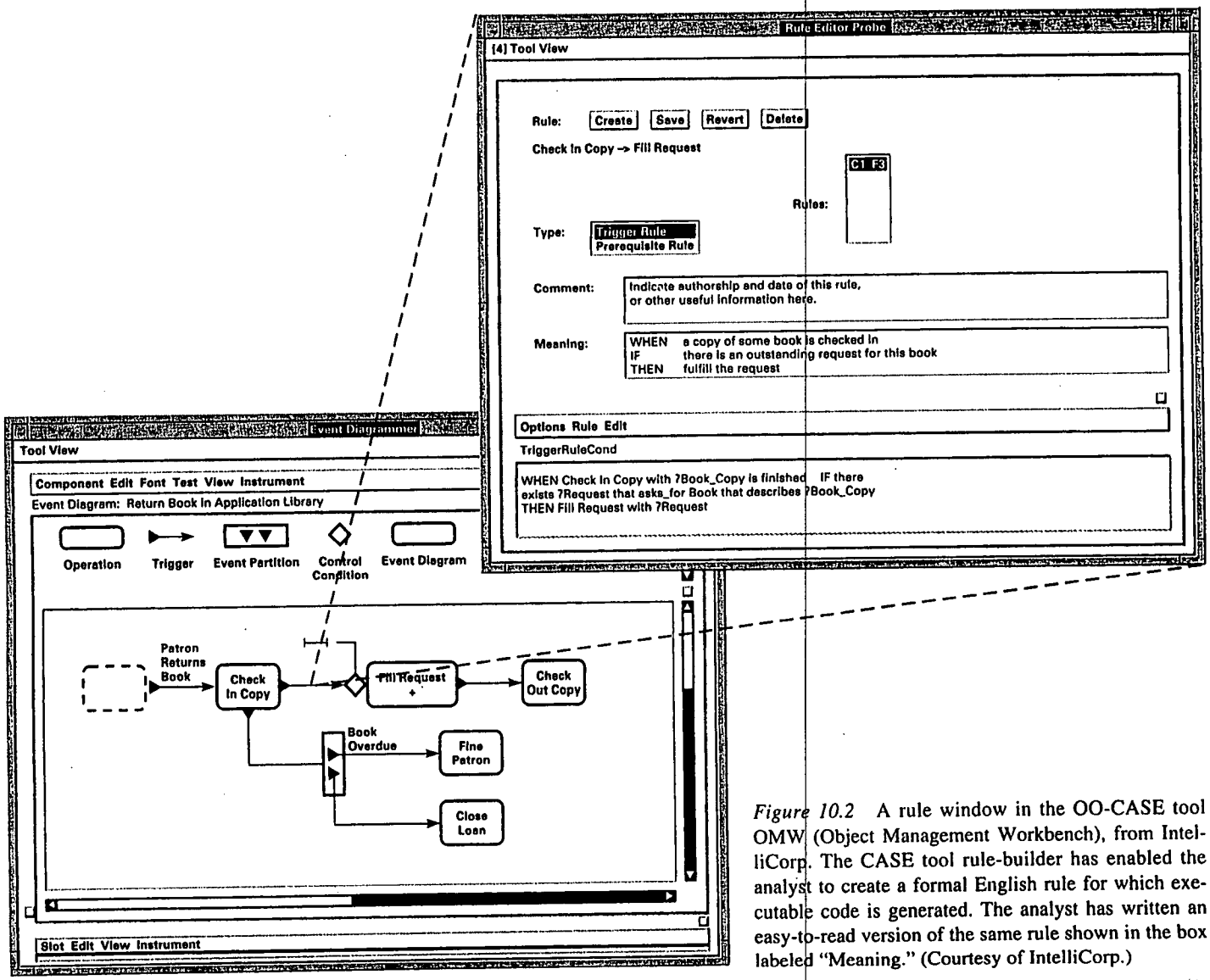


Figure 10.2 A rule window in the OO-CASE tool OMW (Object Management Workbench), from IntelliCorp. The CASE tool rule-builder has enabled the analyst to create a formal English rule for which executable code is generated. The analyst has written an easy-to-read version of the same rule shown in the box labeled "Meaning." (Courtesy of IntelliCorp.)

cutable code is generated. The analyst has written an easy-to-read version of the same rule shown in the box labeled "Meaning." (Courtesy of IntelliCorp.)

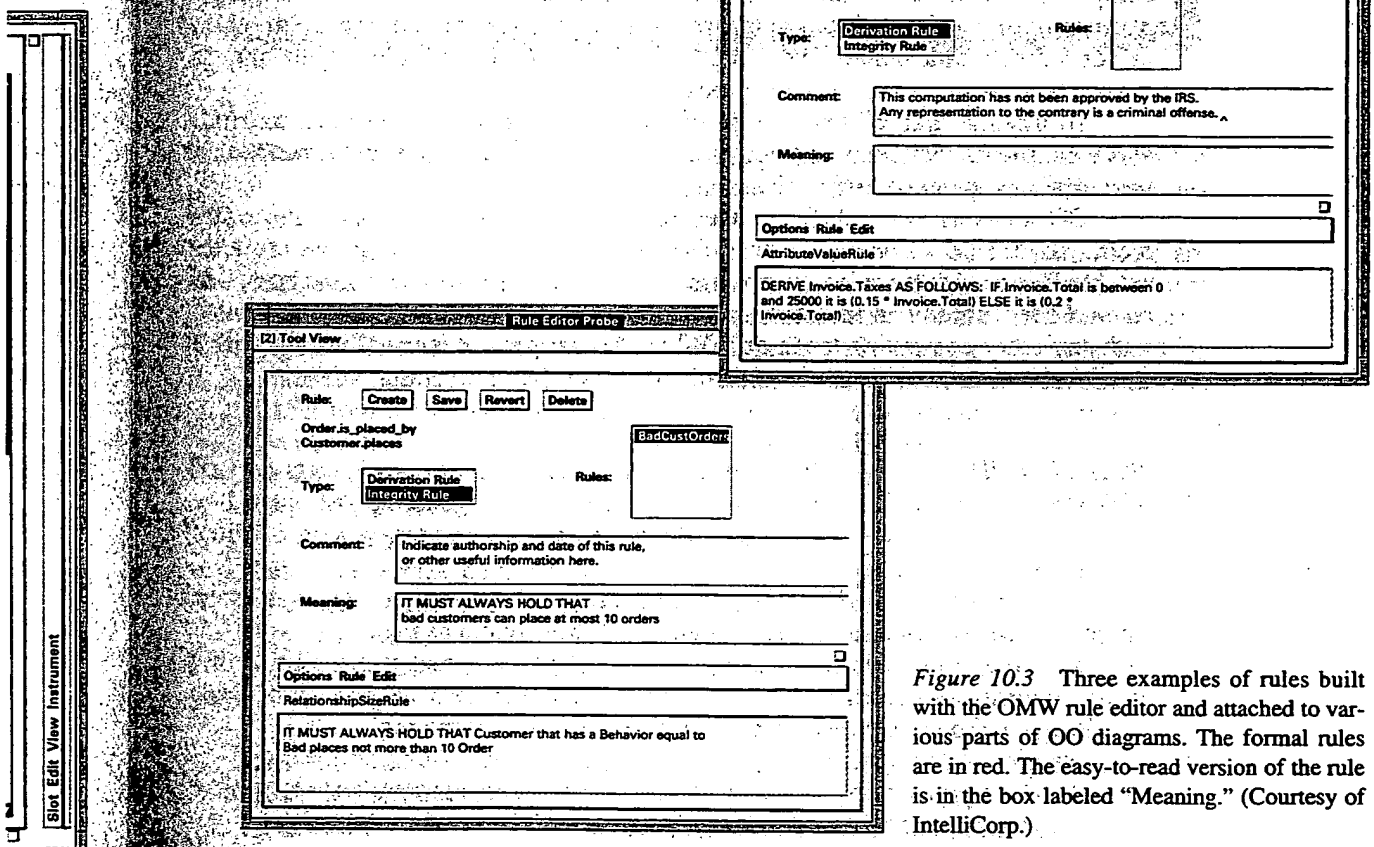


Figure 10.3 Three examples of rules built with the OMW rule editor and attached to various parts of OO diagrams. The formal rules are in red. The easy-to-read version of the rule is in the box labeled "Meaning." (Courtesy of IntelliCorp.)

Just as traditional programs, repeatedly added to, become great rolling snowballs of code, which are murder to maintain, so large expert systems grew into vast collections of rules that became difficult to understand. The answer to both situations is design in an object-oriented fashion where the classes have *methods* that are relatively simple and the overall model is easy to understand.

The artificial-intelligence community developed *frame-based* techniques for designing expert systems and representing knowledge. A *frame* is essentially an object. It has packets of rules associated with it that are used with an inference engine when a request is sent to the frame. Some frame-based software for building expert systems evolved into object-oriented toolkits with methods, inheritance, and encapsulation.

UNNECESSARY RULES

The early expert systems used a large bucket of rules in which the inference engine went fishing for rules to use. Most of the rules in these early systems are unnecessary when object-oriented design is done. They represent information that should be affiliated with OO diagrams, such as object-relationship diagrams, composed-of diagrams, state-transition diagrams, and event diagrams. For example, cardinality information on object-relationship diagrams and the information in normalized data models were represented as production rules. They should have been quite separate from the production rules.

To assume that *all* knowledge should be represented by production rules was a simplistic viewpoint that led to severe code inefficiencies and systems that were difficult to debug and evolve.

OO analysis has a different viewpoint. Rules and other declarative statements should be associated with OO diagrams so that the diagrams are executable. Most of these rules are not used by an inference engine. Sometimes, one *method* does employ an inference engine, just as a method could be implemented with any other computing technique. When this happens, it is usually done with a small number of rules (often less than 100) and so the inferencing is fast.

VISIBLE AND NONVISIBLE RULES

Some rules need to be visible to end users so that they may check them in a workshop (Chapter 14). These should be in English, perhaps with detailed comments.

Other rules are for I.S. professionals and need not be seen by end users. These include certain integrity rules and rules for technical design.

Yet other rules are internal to the OO-CASE tool and the facilities that ensure integrity and consistency in its repository. These rules are part of the basic tool design and need not be directly visible to I.S. users of the tool.

We should distinguish these three categories of rules (Box 10.4).

Box 10.5 gives other examples of rules which business people see as part of the OO enterprise model.

BOX 10.4 Categories of rules.

Nonvisible Rules

Rules internal to repository
Rules internal to OO-CASE toolset

Rules for I.S. Technicians

Rules for technical design
Integrity rules

Rules for End Users

Business policy rules
Derivation rules
Intelligent enterprise model

TRACEABILITY

Each business policy should be directly traceable from the business people who help establish the rules to the code that is generated from them.

Traceability may begin in an end-user workshop in which the policies are established or reviewed with the business people most familiar with the subject area. In such a workshop, each rule is expressed both in formal and easy-to-read English. Comments may be recorded about the rule to state why it is used, to establish its business context, or to indicate when it should be reexamined. A CASE tool for OO analysis ought to record *spoken* comments about rules. Spoken comments can indicate why the rule was established and who was responsible for it. At a later time, the spoken comments might be reviewed and some rules refined or replaced.

To trace the effect of rules, an interpretive code generator is useful that immediately generates code and allows tables to be populated with instances of the objects and their data. This allows an analyst to check with end users in a workshop that the system, or a piece of the system, is working as intended. The interpreted code grows and is refined. Eventually, it will be compiled for efficiency.

CHANGING HOW BUSINESSES ARE RUN

The ability to translate policies directly in executable code has the potential of changing the way businesses are run. Just as a plant controller can turn a valve and

BOX 10.5 Examples of visible rules.

Examples of rules that business people can validate and change. All such rules appear in windows on the OO-CASE diagrams. (From IntelliCorp's OMW.)

Banking

If LiquidityRatio < 1 then Liquidity is bad

LiquidityRatio = ShortTermAssets/ShortTermLiabilities

ShortTermAssets = CashinHand+AccountsReceivable

ShortTermLiabilities = BankLoans+AccountsPayable

When ReviewLoan is completed
    If MonthlyRepayment <= 10% of MaximumRepayment
    then GrantLoan

When ReviewLoan is completed
    If MonthlyRepayment > MaximumRepayment
    then RejectLoan

When ReviewLoan is completed
    If MonthlyRepayment > 50% and <= 100% of MaximumRepayment
    then SeekGuarantees

When ReviewLoan is completed
    If MonthlyRepayment > 10% and < 50% of MaximumRepayment
    then ReviewSituation

Car Rental

On CheckInCar
    MileageUsed = MilesReturned – MilesOut
    GasUsed = 1 – GasGaugeReading
    DateReturned = Today

When CheckInCar is completed then CalculateCharges

On CalculateCharges
    RateCharge = (HireRate)*(DateReturned – DateOut)
    GasCharge = GasUsed*ModelRefuelCharge
    TaxCharge = (RateCharge + GasCharge)*StateTax

BOX 10.5 *(Continued)*

It must hold that DateReturned >= DateRented

HireRate = ModelDailyRate if DateReturned – DateOut < 7

HireRate = ModelWeeklyRate if DateReturned – DateOut >= 7

Credit Card Application

ApplicationStatus = Rejected
    If MonthlyIncome < IncomeThreshold or
    If ApplicantAge < 18

When ReviewApplication is completed
    if ApplicationStatus is Rejected
    then ProcessRejection

When ReviewApplication is completed
    if ApplicationStatus is not Rejected
    then ReviewCreditworthiness

ApplicantPosition = MonthlyIncome – MonthlySpending

If ApplicantPosition is Negative
    then ApplicantCreditworthiness is bad

MonthlyIncome = MonthlySalary + OtherIncome

MonthlySpending = MonthlyHousingCost + OtherSpending

It must hold that an Applicant
    is a PersonalApplicant
    or a CorporateApplicant

When Today = ApplicationReceivedDate + 30 then ResponseOverdue

adjust a manufacturing process, so business people may in the future be able to modify rules of the business and make a direct change in the automated operation of the business. They will adjust and try to optimize how the business is conducted.

REFERENCES

1. Meyer, Bertrand, *Object-Oriented Software Construction,* Prentice Hall, New York, 1988.

2. Van Assche, Franz, *Rule-Based IEM,* James Martin and Co., internal paper, December, 1991.

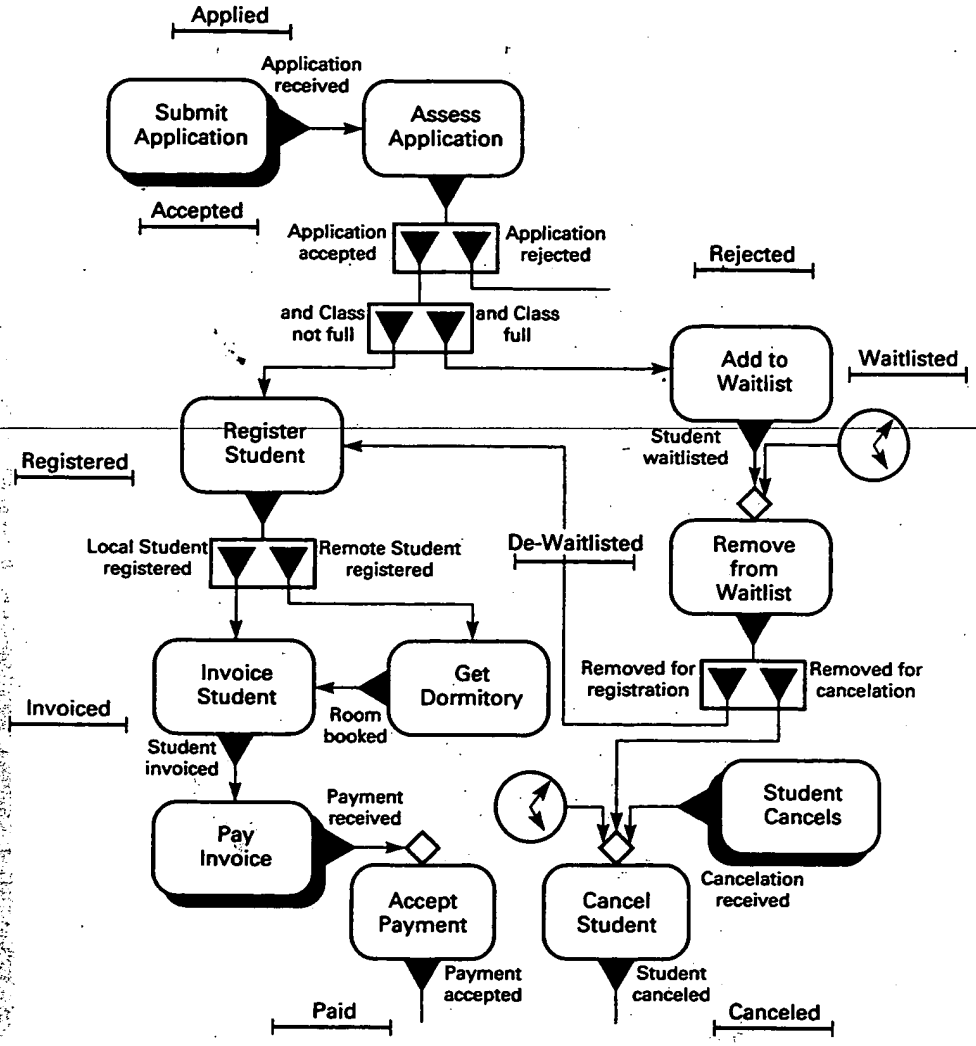


Figure 11.9 An event diagram for the process of registering students. The states of the Student object are shown on this diagram. Figure 11.8, showing the state changes in this diagram, can be generated automatically from this diagram.

State-transition diagrams may be shown as windows that are opened over event diagrams. Figure 11.10 shows two state-transition windows opened on the event diagram of Fig. 11.9, with color showing the state of the Student object type.

Conversely, Fig. 11.11 shows a state-transition diagram and a window containing an event diagram corresponding to one state transition.

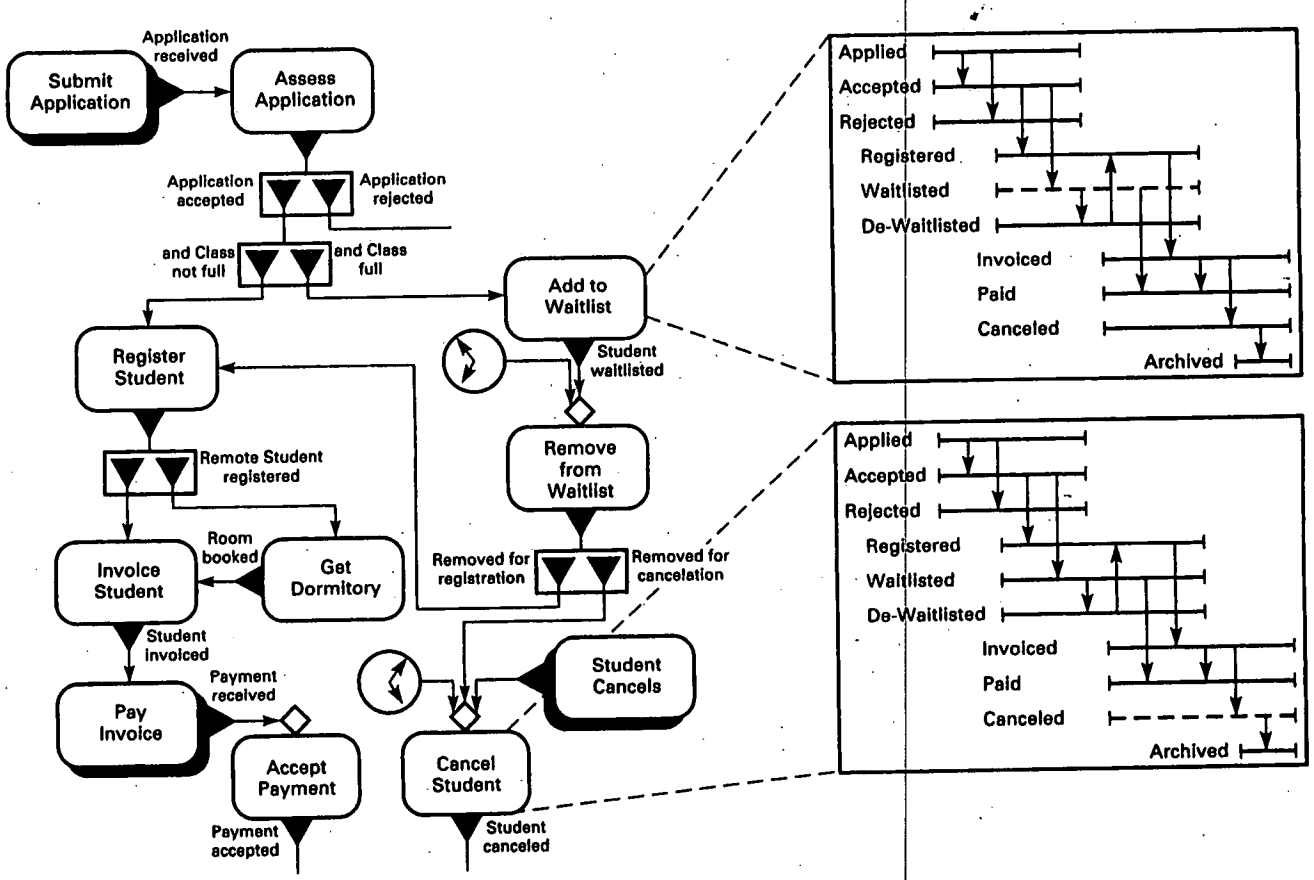


Figure 11.10 Two state-transition windows opened on the event diagram for registering students.

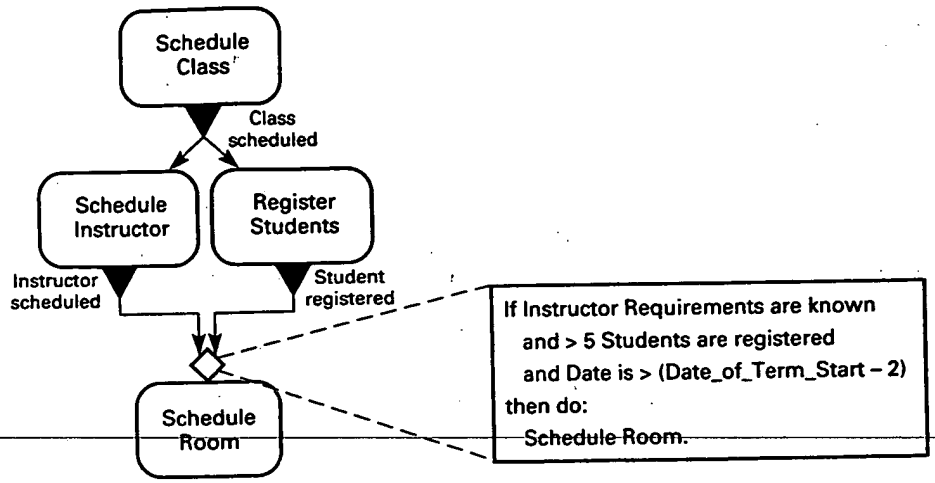


Figure 11.16 A rule connected to the event diagram of Fig. 11.6.

is the kind of service requested, and the method is the specification of the programming code for it.

> An *operation* is a process that can be requested as a unit.
> A *method* is the specification of an operation.

The methods in a class manipulate *only* the data structure of that class. They cannot directly access the data structure of a different class. To use the data structures of a different class, they must send a request to that class. *Encapsulation* must be preserved in OO design.

The designer takes an operation on the event diagram and adds more detail to it in order to create a design. For example, a model produced in analysis may contain the operation Create Purchase Order:

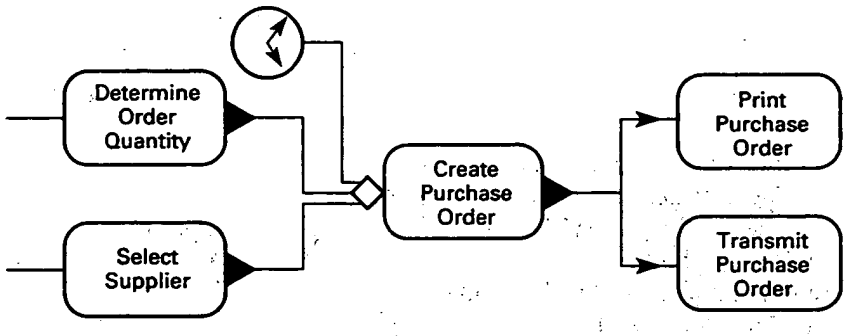
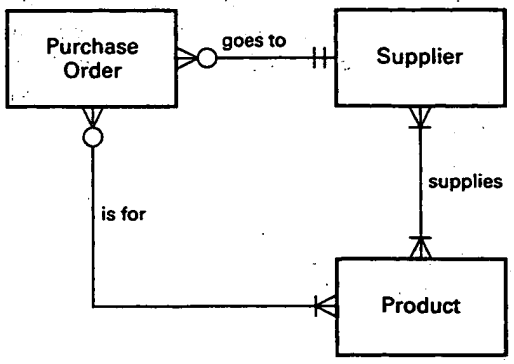


Fig. 12.2.

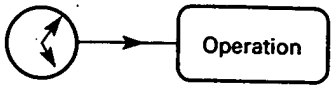
Along with this, Object Structure Analysis has identified object types, such as Supplier, Purchase Order, and Product.



Data modeling has identified the attributes that are required for each of these objects. The attributes are associated with the object identifier in a correctly normalized manner. One object may have multiple normalized groups of attributes (i.e., it contains multiple entity types).

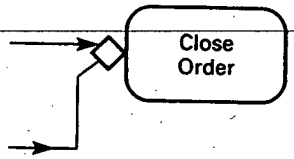
CLOCK EVENTS

A special type of event is a clock time being reached that triggers some operation. This type of event is drawn like a clock face:



CONTROL CONDITIONS

Some operations can only take place when certain conditions apply. These are called *control conditions.* Control conditions are shown with a small diamond at the front of an operation box:



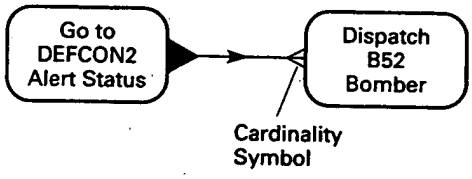
The diamond shape is similar to the decision symbol used in flow charting.

The control-condition diamond can define a single condition or a complex collection of Boolean conditions. The CASE-tool user should be able to mouse-click on the control-condition diamond and display a window showing the conditions.

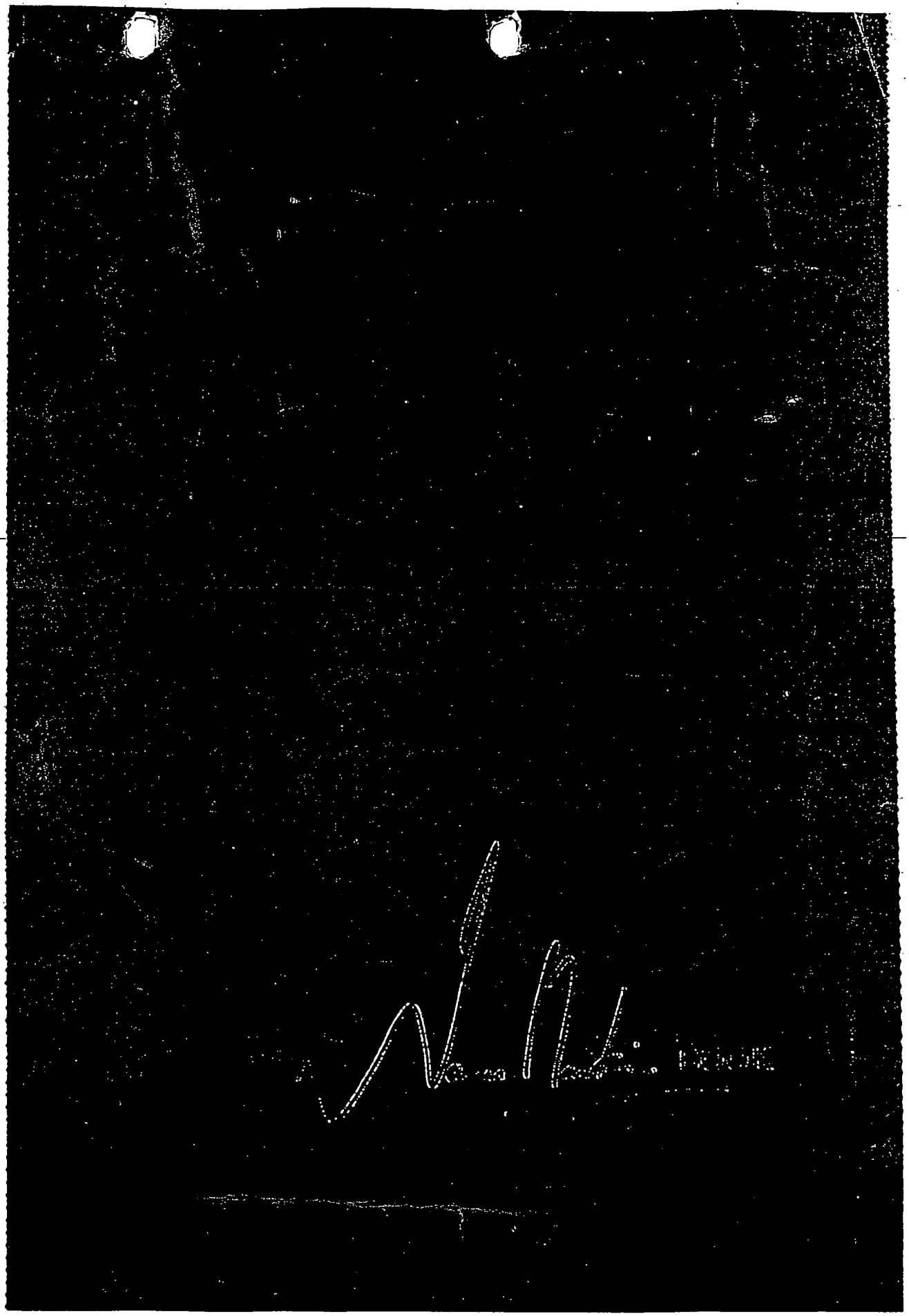
TRIGGERS

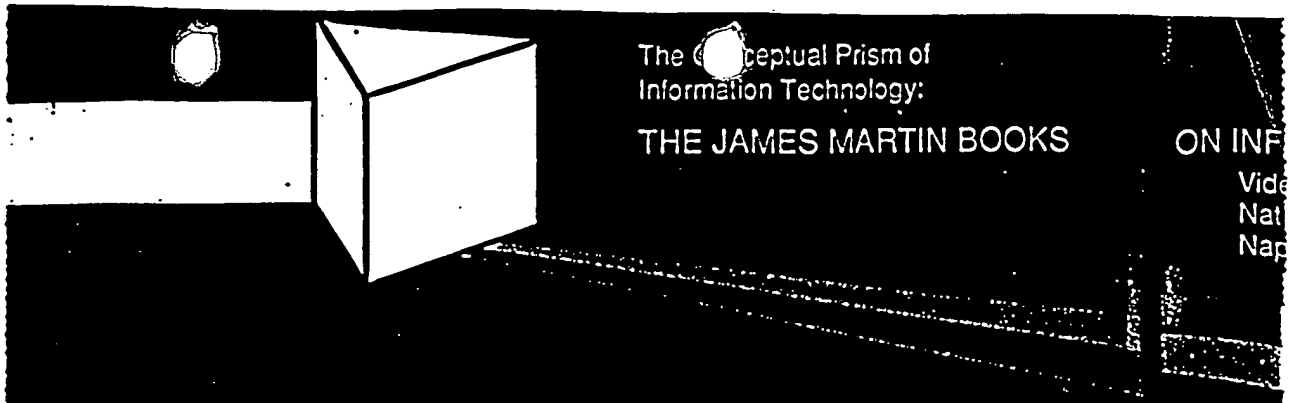
A line going to an operation box indicates that the operation is triggered by the occurrence of a preceding event. The trigger may have a rule associated with it.

An event can trigger multiple instances of the same operation. A one-with-many crow's-foot notation can be used to indicate this:



The trigger-rule line indicates the association of an event type to the operation invoked. Additionally, it can indicate the way in which the necessary objects are supplied to the operation it invokes. In this way, the trigger rule defines a causal relation between event and operation—as well as a form of "data flow."

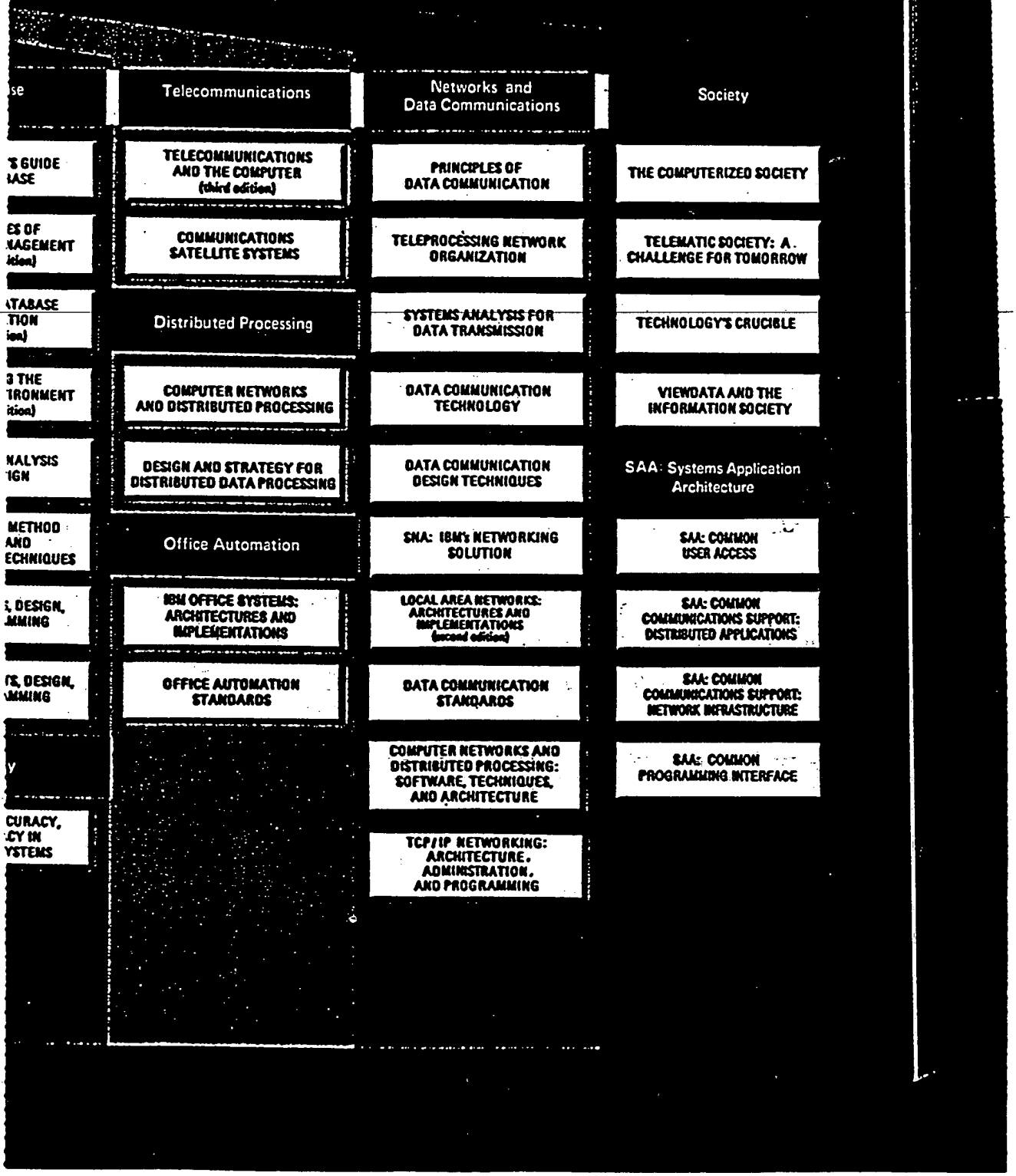




rmation Technology agement and Strategy	Methodologies for Building Systems	Analysis and Design	CASE	Datab
AN INFORMATION SYSTEMS MANIFESTO	STRATEGIC INFORMATION PLANNING METHODOLOGIES (second edition)	STRUCTURED TECHNIQUES: THE BASIS FOR CASE (revised edition)	STRUCTURED TECHNIQUES: THE BASIS FOR CASE (revised edition)	AN END USER TO DATAI
RMATION ENGINEERING (Book I: Introduction)	INFORMATION ENGINEERING (Book I: Introduction)	DATABASE ANALYSIS AND DESIGN	INFORMATION ENGINEERIN (Book I: Introduction)	PRINCIPL DATABASE MA (second ed
RMATION ENGINEERING (Book II: Planning and Analysis)	INFORMATION ENGINEERING (Book II: Planning and Analysis)	DESIGN OF MAN-COMPUTER DIALOGUES	Languages and Programmi	COMPUTER DA ORGANIZA (third edi
ATEGIC INFORMATION NING METHODOLOGIES (second edition)	INFORMATION ENGINEERING (Book III: Design and Construction)	DESIGN OF REAL-TIME COMPUTER SYSTEMS	APPLICATION DEVELOPMEN WITHOUT PROGRAMMERS	MANAGING DATABASE ENVI (second edi
TWARE MAINTENANCE: THE PROBLEM AND ITS SOLUTIONS	STRUCTURED TECHNIQUES: THE BASIS FOR CASE (revised edition)	DATA COMMUNICATIONS DESIGN TECHNIQUES	FOURTH-GENERATION LANGUAGES (Volume I: Principles)	DATABASE AN AND DESI
ESIGN AND STRATEGY FOR DISTRIBUTED DATA PROCESSING	Object-Oriented Programming	DESIGN AND STRATEGY FOR DISTRIBUTED DATA PROCESSING	FOURTH-GENERATION LANGUAGES (Volume II: Representative 4GL	VSAM: ACCESS I SERVICES A ROGRAMMING TE
Expert Systems	OBJECT-ORIENTED ANALYSIS AND DESIGN	SOFTWARE MAINTENANCE: THE PROBLEM AND ITS SOLUTIONS	FOURTH-GENERATION LANGUAGES (Volume III: 4GLs from IBM)	DB2: CONCEPTS AND PROGRAM
LDING EXPERT SYSTEMS: A TUTORIAL	PRINCIPLES OF OBJECT-ORIENTED ANALYSIS AND DESIGN	SYSTEM DESIGN FROM PROVABLY CORRECT CONSTRUCTS	Diagramming Technique	DMS/R: CONCEPTS AND PROGRAM
	OBJECT-ORIENTED METHODS: A FOUNDATION	INFORMATION ENGINEERING (Book II: Planning and Analysis)	DIAGRAMMING TECHNIQUE FOR ANALYSTS AND PROGRAMMERS	Security
	OBJECT-ORIENTED METHODS: THE PRAGMATICS	INFORMATION ENGINEERING (Book III: Design and Construction)	RECOMMENDED DIAGRAMMI STANDARDS FOR ANALYST AND PROGRAMMERS	SECURITY, ACC AND PRIVAC COMPUTER SY
	OBJECT-ORIENTED TOOLS		ACTION DIAGRAMS: CLEAR STRUCTURED SPECIFICATIO PROGRAMS, AND PROCEDUR (second edition)	

ORMATION TECHNOLOGY

o education courses are available on these topics through
onal Education Training Group, 1751 West Diehl Road,
erville, IL 60563-9099 (tel: 800-526-0452 or 708-369-3000).



RINCIPLES
)F OBJECT-ORIENTED
.NALYSIS AND DESIGN

A James Martin BOOK

PRINCIPLES OF OBJECT-ORIENTED ANALYSIS AND DESIGN

AMES MARTIN

P T R PRENTICE HALL
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data
Martin, James (date)
Principles of object-oriented analysis and design / by James Martin.
p. cm.
"A James Martin book."
Includes bibliographical references and index.
ISBN 0-13-720871-5
1. Object-oriented programming (Computer science) I. Title.
QA76.64.M38 1993
005.1'1—dc20 92-31822
CIP

TITL: Principles of object-oriented analysis and design / aJames Martin.
PHYS: 412 p.
CLNA: Prentice-Hall, Inc., a Simon & Schuster Company
INNN: notice: James Martin and cJames I. Odell
DCRE: 1992 DPUB: 29Oct92 DREG: 22Dec92
LINM: NM: "new, updated, and revised material."
ECIF: 1/B/D

Editorial/production supervision: *Kathryn Gollin Marshak*
Liaison: *Mary P. Rottino*
Jacket design: *Lundgren Graphics*
Pre-press buyer: *Mary Elizabeth McCartney*
Manufacturing buyer: *Susan Brunke*

Published by P T R Prentice Hall
Prentice-Hall, Inc.
A Paramount Communications Company
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:

Corporate Sales Department
P T R Prentice Hall
113 Sylvan Avenue
Englewood Cliffs, NJ 07632
Phone 201-592-2863; Fax 201-592-2249

Printed in the United States of America

10 9 8 7 6 5 4

ISBN 0-13-720871-5

ISBN 0-13-720871-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

THE JAMES MARTIN BOOKS
currently available from Prentice Hall

- Application Development Without Programmers
- Building Expert Systems
- Communications Satellite Systems
- Computer Data-Base Organization, Second Edition
- Computer Networks and Distributed Processing: Software, Techniques, and Architecture
- Data Communication Technology
- DB2: Concepts, Design, and Programming
- Design and Strategy of Distributed Data Processing
- An End User's Guide to Data Base
- Fourth-Generation Languages, Volume I: Principles
- Fourth-Generation Languages, Volume II: Representative 4GLs
- Fourth-Generation Languages, Volume III: 4GLs from IBM
- Future Developments in Telecommunications, Second Edition
- Hyperdocuments and How to Create Them
- IBM Office Systems: Architectures and Implementations
- IDMS/R: Concepts, Design, and Programming
- Information Engineering, Book I: Introduction and Principles
- Information Engineering, Book II: Planning and Analysis
- Information Engineering, Book III: Design and Construction
- An Information Systems Manifesto
- Local Area Networks: Architectures and Implementations
- Managing the Data-Base Environment
- Object-Oriented Analysis and Design
- Principles of Data-Base Management
- Principles of Data Communication
- Principles of Object-Oriented Analysis and Design
- Recommended Diagramming Standards for Analysts and Programmers
- SNA: IBM's Networking Solution
- Strategic Information Planning Methodologies, Second Edition
- System Design from Provably Correct Constructs
- Systems Analysis for Data Transmission
- Systems Application Architecture: Common User Access
- Systems Application Architecture: Common Communications Support: Distributed Applications
- Systems Application Architecture: Common Communications Support: Network Infrastructure
- Systems Application Architecture: Common Programming Interface
- Technology's Crucible
- Telecommunications and the Computer, Third Edition
- Telematic Society: A Challenge for Tomorrow
- VSAM: Access Method Services and Programming Techniques

with Carma McClure

- Action Diagrams: Clearly Structured Specifications, Programs, and Procedures, Second Edition
- Diagramming Techniques for Analysts and Programmers
- Software Maintenance: The Problem and Its Solutions
- Structured Techniques: The Basis for CASE, Revised Edition

Figure 17.1 illustrates an example of the ADT named Employee. At its heart, the ADT is defined by its data structure representation. For Employee, this includes data about exemptions, position, salary amount, phone extension, and so on. The ADT is also defined by a set of permissible operations. These operations—such as hire, promote, and change phone extension—provide a suit of armor that protects the underlying Employee structure from arbitrary and unintended use. In other words, the Employee operations provide the only method of accessing and maintaining the data of an Employee.

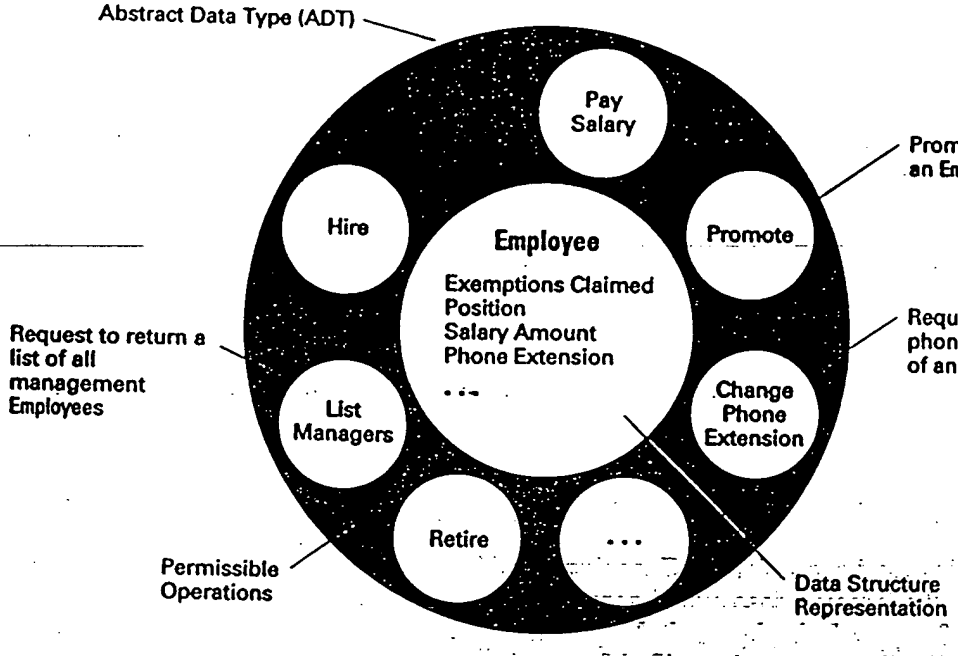


Figure 17.1 Objects are accessible only by named operations; all else is hidden.

Additionally, each *method* or processing algorithm, employed to carry out an operation, is hidden from its users. What the user must provide is an appropriate object to *request* the operation along with any applicable supporting parameters. For example, Fig. 17.2 depicts three instances, or *objects*, of Employee. In order to give Bob a promotion, the request must specify Bob, the promote operation, and the salary grade of his promotion. In its abbreviated form, this request could be written: Bob, promote, director.

Objects as Encapsulations

Figure 17.2 depicts three Employee objects as specific representations of the Employee ADT. In this way, an object can be regarded as any instance of an abstract data type—each encapsulating its own private data and its own permissi-

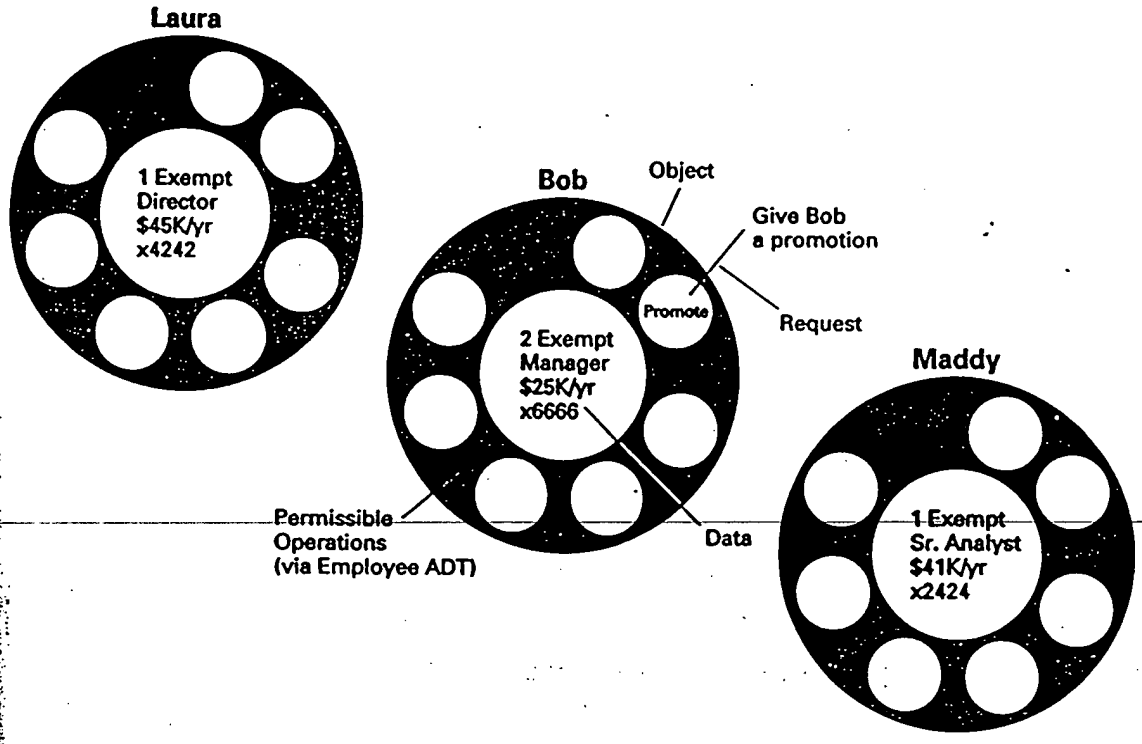


Figure 17.2 An object's permissible operations are defined by its ADT.

ble operations. Each object can be considered as a thing in its own right, with its own behavior. While each object should encapsulate a physical record of its own data, encapsulating a physical copy of each operational method is unnecessary and wasteful. Since the same coding is contained within each ADT, an ADT's operations need only be *virtually* available to an object. In other words, all operations that apply to a particular ADT also apply to the instances of that ADT. As illustrated in Fig. 17.3, since Bob is an instance of the Employee ADT, all Employee operations (such as promote) also apply to Bob—without the object having to contain them physically. When an object is an instance of an ADT, this linkage is established. Most ADT-oriented languages accomplish this with a physical pointer mechanism.

OBJECTS AND REQUESTS

With encapsulation, each object need only know *what* it can request of another object, because operations are the *public* interface for all objects. All the specifics of *how* its structure is stored and *how* its operations are coded are tucked neatly out of sight. This not only protects each object, it simplifies the

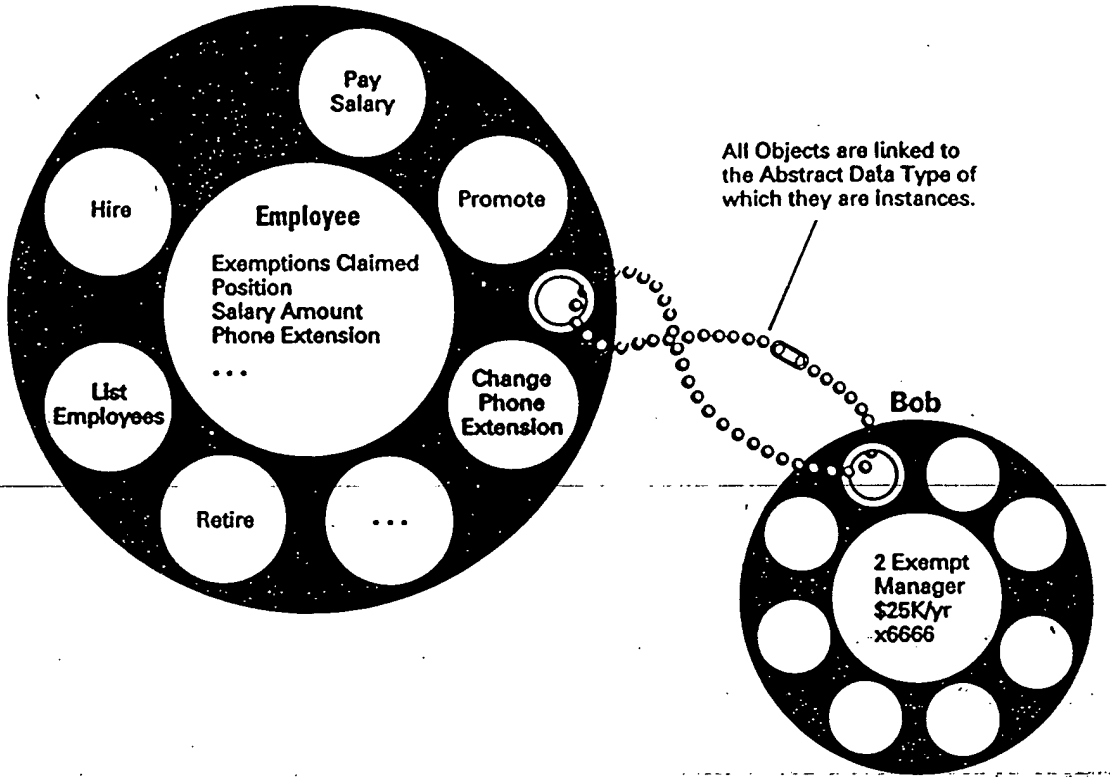


Figure 17.3 Every ADT is linked to its instances.

interactions among objects. Most OO languages call these interactions *messages*. For instance, a Customer object can send a message to an Order object to add a product to its already existing line items. The Order object, in turn, may send a message to a Customer Account object with a request to update the amount due for the customer.

The standard term emerging for a message is a *request* (Fig. 17.4). A request is a more general notion, because more than one object can participate in a request. For instance, to which object is a message sent so that a Part can be placed in a Bin: the Part or the Bin? A request involves both by specifying the Part and Bin objects as parameters along with the operation name. It then lets the method selection mechanism locate the appropriate method for placing inventory. In short, requests expand the notion of message by indicating: With *these* objects, do this.

INHERITANCE AND POLYMORPHISM

Inheritance is an important feature of OO design. While different OO programming languages have different inheritance mechanisms, we can think about

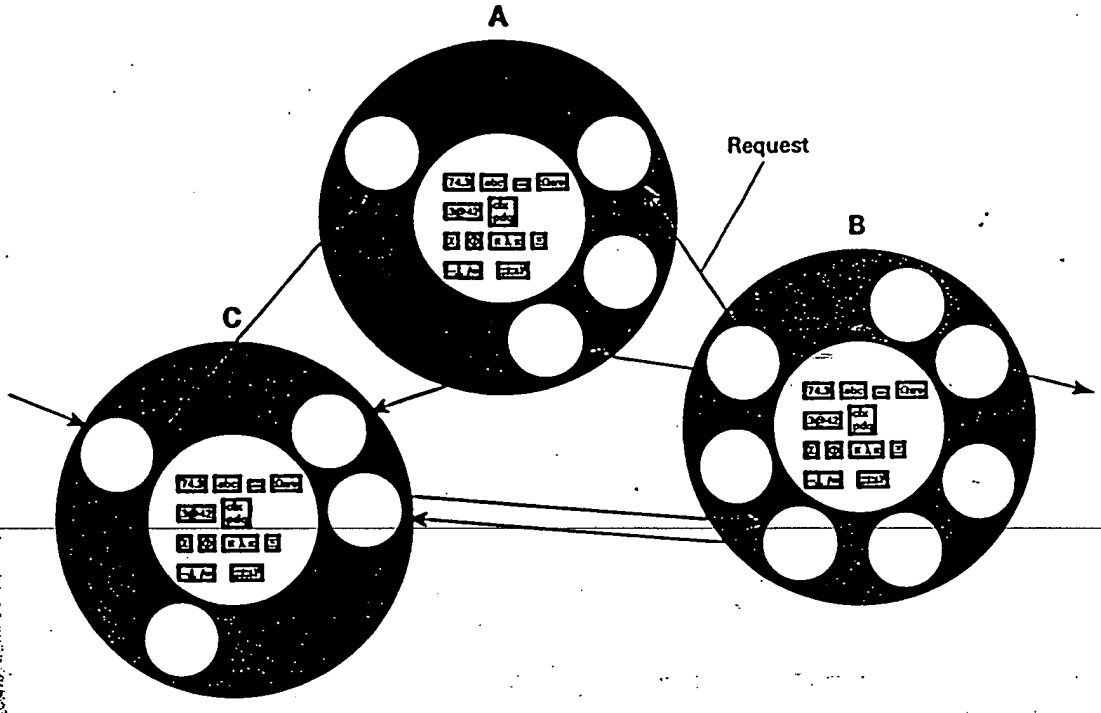


Figure 17.4 The operation of one object *requests* the operation of another.

them in the following way. When a request for an operation goes to a subclass, the list of permissible operations of that class is checked. If the operation is found on the list, it is invoked; if not, the parent classes are examined to locate the operation.

An important feature of inheritance is the ability of a class to *override* inherited features. Here, the processing algorithm, or *method,* of an inherited operation can be redefined at the subtype level. The example in Fig. 17.5 illustrates three classes. The most general, Polygon, contains the data structure and permissible operations for polygons. Because every instance of a Rectangle is also an instance of a Polygon, the Rectangle subtype need not repeat those features it inherits from Polygon. However, while all Polygon operations apply to subtypes, the *method* of operation may be different. For example, the method of rotating a Polygon is the same as rotating a Rectangle. However, the method of computing perimeters may differ. The perimeter of a Polygon is the sum of all its sides; the perimeter of a Rectangle is the sum of two of its adjacent sides multiplied by two. The Square perimeter differs again as the product of multiplying four times the length of any one side.

Whenever a request is made for an operation on an object, the method selected depends on whether or not the inheritance hierarchy has been overrid-

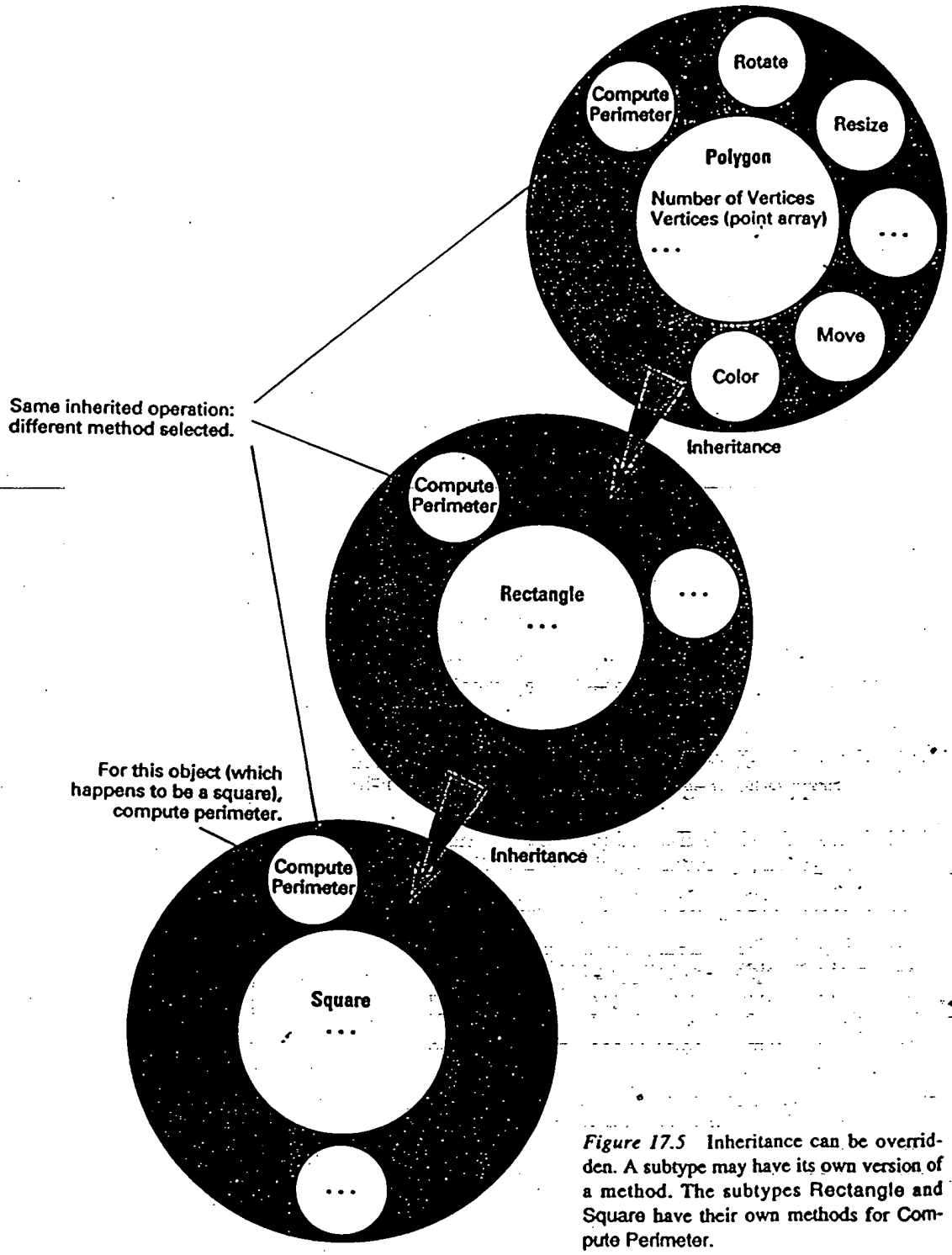


Figure 17.5 Inheritance can be overridden. A subtype may have its own version of a method. The subtypes Rectangle and Square have their own methods for Compute Perimeter.

den. The method for moving a Square three centimeters to the right is selected from Polygon. However, even though the *operation* for compute perimeter is inherited from the Polygon, the *method* selected for a Square is located in the Square class. This approach to overriding inheritance properties through redefinition is often referred to as a category of *polymorphism* known as *parametric polymorphism* [2]. An *operation*, then, is the kind of process being requested. Its *method* is the specification of how to carry out the operation.

CANCELING INHERITED FEATURES

Some expert systems in AI allow some inherited features to be canceled. For example, one of the features of Birds includes flying. However, this feature does not apply to Penguins and Ostriches. Therefore, this feature can be canceled. The arbitrary overriding and canceling of inherited features is a questionable practice. Logically it is incorrect, because, by definition, all features of a type apply to its subtypes. Therefore, to rectify the problem of Birds flying and Penguins not, the subtyping hierarchy needs to be changed. To solve this, Bird can be specialized into two subtypes: Flying Bird and Nonflying Bird. Following this, all the data structures and operations relating to flying should be shifted from Bird to Flying Bird. Types such as Penguin and Ostrich should then be realigned as subtypes of Nonflying Bird. This would correct the logical inconsistency. However, *physically* it might create an intolerable system overhead. For this reason, some languages allow the programmer to deviate from what is logically correct—for the sake of performance.

CHARACTERISTICS OF OO LANGUAGES

To be described as an OO programming language, a language must support

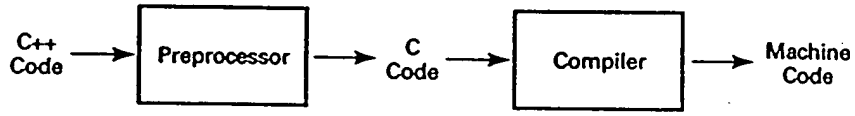
- *Classes and encapsulation.* Each class has certain kinds of data. Each class protects its data from improper use by offering a number of permissible operations. To accomplish this, both the data representation and its permissible operations are veiled with a protective covering that hides the details of its implementation.
- *Method selection.* With method selection, the user need only specify which operation should be applied to an object (or in more expanded languages, one or more objects). The system will then choose the method appropriate for the specified parameters. In other words, the user only needs to specify what is to be done, and the method selector determines how it is to be applied. Polymorphism is one of the common applications of method selection.
- *Inheritance.* Classes inherit the data types and methods of higher level classes. Inheritance allows systems to be constructed from existing class hierarchies. It provides mechanisms for both construction and reuse of software. In this way, we do not have to reinvent the wheel—only the portion of it that is different. Inheritance imposes a mechanism on classes that greatly reduces the complexity of the resulting systems.

Some OO languages support multiple inheritance; others support inheritance from only one parent class.

PURE VERSUS HYBRID OO LANGUAGES

Some OO languages have been designed specifically for OO programming with objects, encapsulation, method selection, and inheritance. Smalltalk was the first language developed purely for OO programming. Following Smalltalk, Actor and Eiffel evolved as *pure* OO languages.

Other languages used for OO programming are traditional languages that have had the capability added to them to handle objects, method selection, and inheritance. These are referred to as *hybrid* languages. The preeminent hybrid language C++, an extension of C, is currently the most commonly used language for OO programming. C++ uses a preprocessor that converts C++ code into C code, ready for compilation.



Following the lead of C++, various traditional languages have been adapted for OO programming. Object PASCAL evolved from PASCAL, CLOS from LISP, Objective-C from C, and Object COBOL from COBOL. This evolution of languages is shown in Fig. 17.6.

Traditional compilers can be used with hybrid languages by means of a preprocessor:

An advantage of adding OO capability to an existing language is that the users of that language learn an extension of what they already know, rather than a completely new language.

This evolutionary learning path, however, has a severe disadvantage. OO thinking is very different from traditional structured thinking, and hybrid languages encourage programmers to think in the way they are used to thinking. The programmers then tend to do traditional functional decomposition and use structure charts rather than think in terms of classes and inheritance. They tend to think in terms of data-flow diagrams rather than event diagrams. Many C++ programmers are using procedural code instead of OO methods, requests, and inheritance. They do not convert fully to the OO mindset.

Often, the best way to make a C programmer learn to use C++ in an OO fashion is to make him program in Smalltalk for six months. The problem with

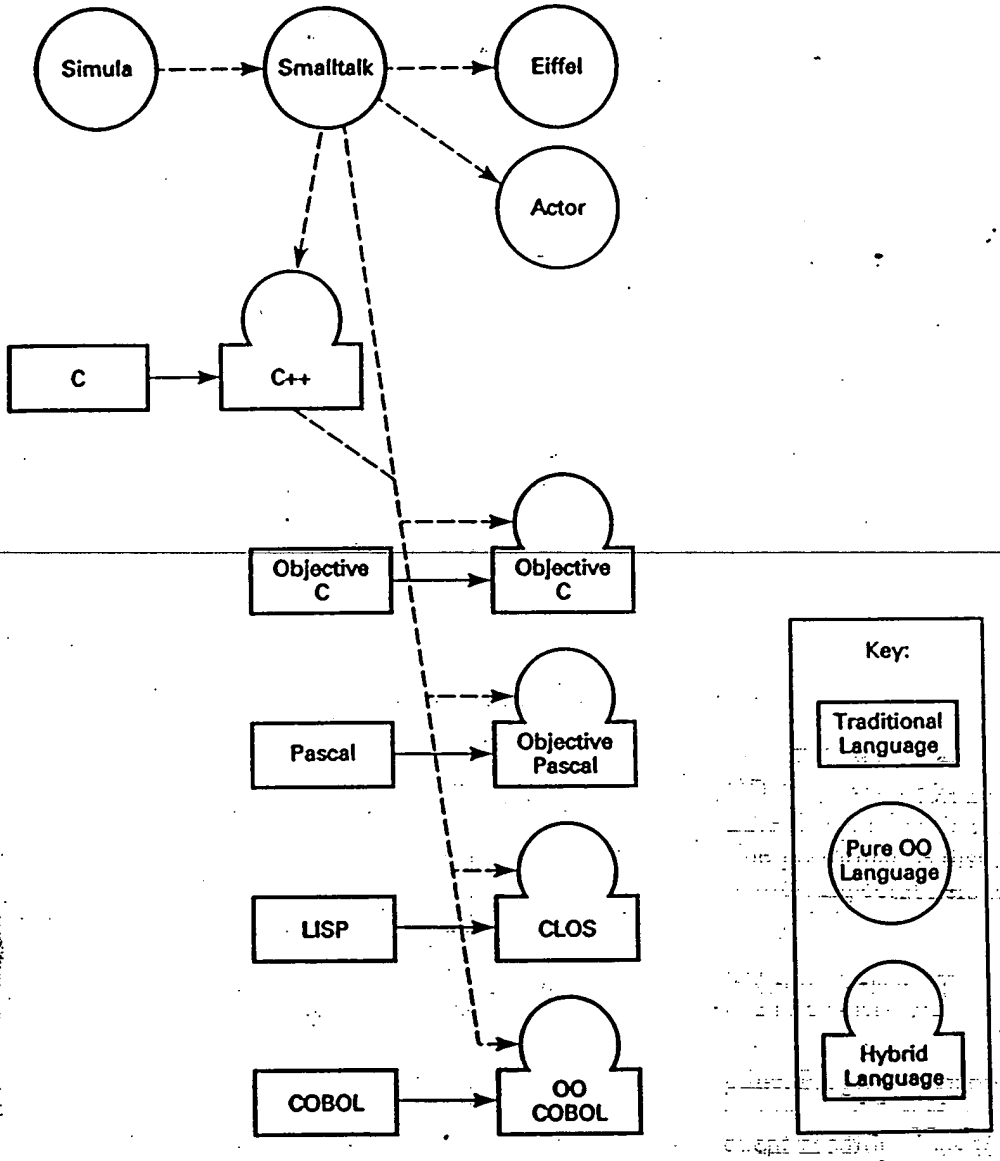


Figure 17.6 The evolution of languages for OO programming.

this is that he might not want to leave Smalltalk, which often stimulates creativity better than C++.

C++ has been referred to as a horseless carriage. The first automobiles were called horseless carriages because they looked like traditional carriages with the horse removed and a motor added. Because hybrid OO languages look like tradi-

tional languages, they tend to encourage traditional thinking—which is difficult to unlearn.

For a *new* programmer, the pure OO languages are easier to learn than the hybrid languages. Smalltalk is fairly simple and uses English-like constructs. New programmers tend to learn it quickly. However, C++ is particularly difficult to learn because it requires two stages of learning. The programmer must first learn C, which is a low-level language originally created for programming operating systems, and then the OO additions to C, which are unnecessarily difficult.

Pure OO languages, used well, have several advantages over traditional programming:

- easier learning
- reduction of complexity (a McCabe metric of 3 rather than 10)
- easier debugging
- reusable classes
- reusability from inheritance
- complexities hidden by encapsulation
- easier to make changes
- hence greater creativity

This increased power and flexibility for the programmer has a price. Pure OO languages often give machine performance lower than hybrid languages. Like C, C++ is a fast, compact language, finely tuned for high-speed execution. The performance disadvantage is steadily being overcome with better design of optimizing compilers for pure OO languages.

ENFORCEMENT OF DISCIPLINE

OO advantages are achieved because of the discipline associated with encapsulation and inheritance. Hybrid languages make it only too easy to avoid this discipline. Programmers often bypass encapsulation in favor of a quick solution. This can cause debugging problems and make it much more difficult to make subsequent modifications to programs. It generally lowers the quality and reusability of the code.

> Enforcement of OO principles, with their substantial benefits, is an argument for using *pure*, rather than hybrid, OO languages. Similarly, at a higher level, it is an argument for using pure, rather than hybrid CASE tools.

INTERPRETERS VERSUS COMPILERS

An OO language should be interpretive, which enables the programmer to run the code as soon as he creates it. As he makes minor changes, he can immediately run them without waiting for the entire program to be relinked. This immediate running of changes enables the programmer to be more creative—quickly catching errors and observing the effect of changes.

Imagine a sculptor not being able to see changes he makes to his sculpture until some time later in the day when it is "compiled." This would drastically reduce the sculptor's ability to be creative. In a similar way, we want to increase the ability of the programmer to be creative.

A problem with interpreters is that they generally give worse machine performance than compilers. Optimizing compilers take multiple passes through the code linking it in such a way as to achieve the best machine performance. However, languages using conventional compilers limit creativity, because each small change requires relinking the whole program which takes from minutes to hours.

Some OO languages solve this problem by means of a *dynamic compiler*. This compiles a *method* (rather than compiling entire programs) whenever the method is encountered. It keeps a pool of recently compiled methods in the main memory to avoid recompilation whenever possible. This is a compromise between traditional interpreters and compilers. The programmer can run changes when he makes them, and the efficiency of runtime execution is high (Fig. 17.7).

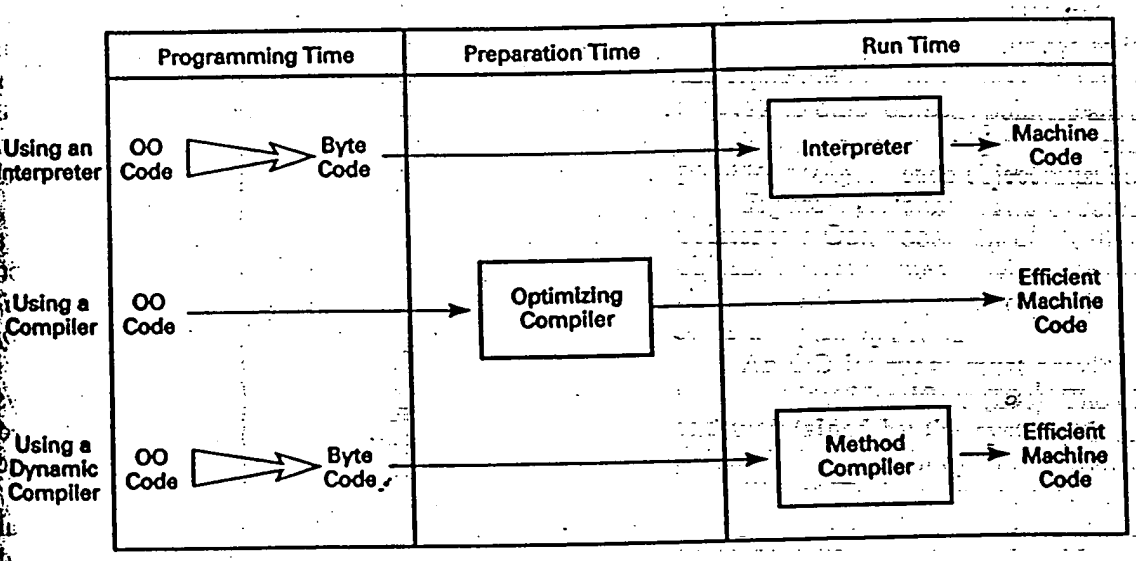


Figure 17.7 To maximize programmer creativity, the programmer should be able to execute changes as soon as he makes them—with no delay between programming time and runtime. This is achieved by using an interpreter or dynamic compiler.

To allow the programmer to interact immediately with evolving code and to achieve the best machine performance, a language should have both an interpreter for use during development and a compiler for use when the program is run (Fig. 17.8).

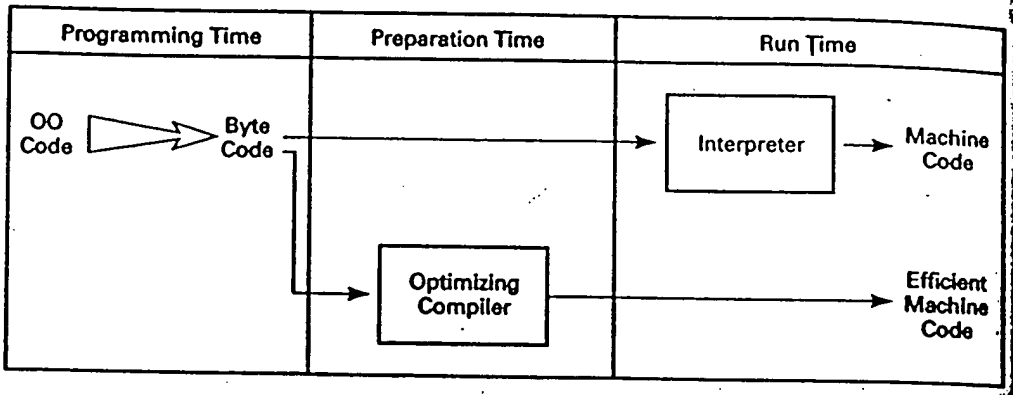


Figure 17.8 To maximize programmer productivity and machine performance, an OO language may have both an interpreter and an optimizing compiler.

POINTERS

OO software uses many pointers. Each object is linked to its class with a pointer mechanism. Finding the right method in a class hierarchy needs pointers. When an object sends requests to other objects, pointers are used.

The *fields* in an object may contain pointers to other objects. In order to point to an object, each object must be uniquely identifiable.

Figure 17.9 shows two Musical Composition objects. Each contains a pointer to a Composer object, Beethoven in this case. The Musical Composition objects have attributes to which it also points, such as *opus number* and *composition name;* the Composer object has attributes, such as *composer name, year of birth,* and *year of death.*

An OO language must provide a good pointer mechanism. This employs unique object identifiers (IDs). The object IDs should be automatically assigned and maintained by the system; the system ensures that they are unique. The pointer links are built by the software either at compilation time or runtime.

Object IDs have important advantages. They are small and require minimal storage. They are much smaller than human-readable names or references based on content. The pointer to the object can be followed quickly. The object can be located with tables, regardless of where it is stored. The ID is independent of the object content. Every variable in the object can be changed and the pointer still points to the correct object.

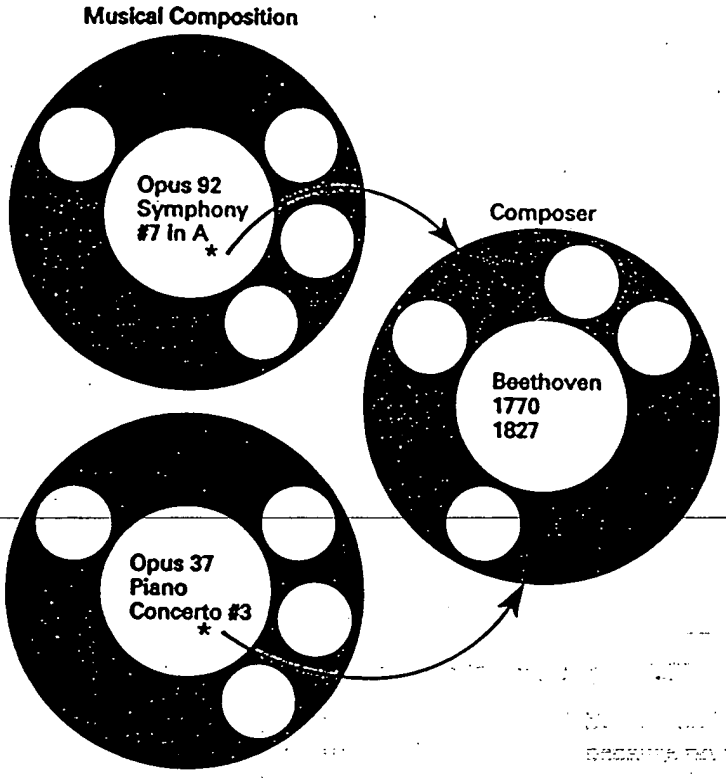


Figure 17.9 Objects are automatically allocated unique identifiers (IDs) so that points can link to them.

STATIC VERSUS DYNAMIC BINDING

Pointers are essential to OO operation. The process of determining the item that is pointed to is called *binding*. Binding identifies the receiver of a request and can be done when either the program is compiled or the program is run. The two categories of binding can be elaborated:

Binding done when the program is compiled is called	*Binding done when the program is run* is called
• compiletime binding,	• runtime binding,
• early binding, or	• late binding, or
• static binding.	• dynamic binding.

Runtime, or dynamic, binding requires somewhat more overhead when the program runs, but it makes *polymorphism* possible. Because of this, some authorities regard it as an essential component of OO systems.

An object may send a request to another object telling it to do something. The receiving object may have its own way of implementing the method requested. It may use a method that did not exist at the time the program was compiled. For example, it might print a chart, but the printer has been changed. It might compute the value of a portfolio with a method that has been modified. The request might go to an object in a distant machine, perhaps even in a different corporation. It may not be practical to recompile all the linkages between objects in different locations each time changes are made, so runtime (or dynamic) binding is used. This gives a high level of flexibility.

Each object knows about its own way of executing a method, but different objects may execute that method in different ways. This is *polymorphism*. The request to Compute Invoice Total might be done differently in different locations because of particular discount schemes or sales, but a request of the same format is sent to all locations. The discounts and sales incentives change so often that it would not be practical to compile all linkages whenever a change occurs. So, runtime binding is used.

Because overhead is associated with runtime binding, some languages automatically use compile-time (or static) binding unless runtime binding is specified. Some versions of C++ permit runtime binding only within one class hierarchy, because polymorphism can usually be confined to one class hierarchy.

AVOIDANCE OF CASE STATEMENTS

Procedural languages have statements (called CASE statements) that allow multiple options. For example

```
IF PRINTER IS TYPE-A DO
   -------
IF PRINTER IS TYPE-B DO
   -------
IF PRINTER IS TYPE-C DO
   -------
IF PRINTER IF TYPE-D DO
   -------
```

CASE statements such as these can usually be replaced by one statement: PRINT. PRINT refers to a method that will be implemented differently in different objects depending on the type of printer used. A new type of printer can then be added and the same request remains valid.

Sometimes CASE statements are lengthy because many possible options are specified, for example a different option for each type of customer. The CASE statement may be repeated in many programs. All options are *hardcoded* into the

program. To add a new option or delete an old one, a change has to be made wherever the CASE statement occurs. A large system may have thousands of CASE statements, hardcoding decisions into its programs. Changing the CASE options requires all the programs to be changed and tested. In OO systems, the hardcoding of options can often be avoided. Each object receiving a request has its own method of implementing that request. New objects with different variations on the theme can be added without changing a request or the code that sends it. This flexible use of methods and polymorphism makes systems much easier to change.

The more complex the system, and the more frequently changes are desired, the greater the advantage of not hardcoding options into the programs. The maintenance of commercial systems would be much easier if they had been built with OO techniques. New objects or new options could be added easily when needed without changing existing objects. This increased flexibility will likely lead to a much higher rate of change in commercial systems.

LANGUAGES AND ENVIRONMENTS

Some OO languages have merely a compiler or interpreter (or both), others have a language development environment, and yet others have a CASE environment (discussed in the following chapter). These facilities are summarized in Fig. 17.10.

REFERENCES

1. Budd, Timothy, *A Little Smalltalk*, Addison-Wesley, Reading, MA, 1987.
2. Cardelli, Luca and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, 17:4, December 1985, pp. 471-522.
3. Cox, Brad J. and Andrew J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach*, (2nd edition), Addison-Wesley, Reading, MA, 1991.
4. Dahl, Ole-Johan and Kristen Nygaard, "SIMULA - An Algol-based Simulation Language," *Communications of the ACM*, 9:9, Sept. 1966, pp. 671-678.
5. Horowitz, Ellis, *Fundamentals of Programming Languages* (2nd edition), W. H. Freeman, New York, 1984.
6. Goldberg, Adele and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
7. Kay, Alan C., "Microelectronics and the Personal Computer," *Scientific American*, September 1977, pp. 231-244.

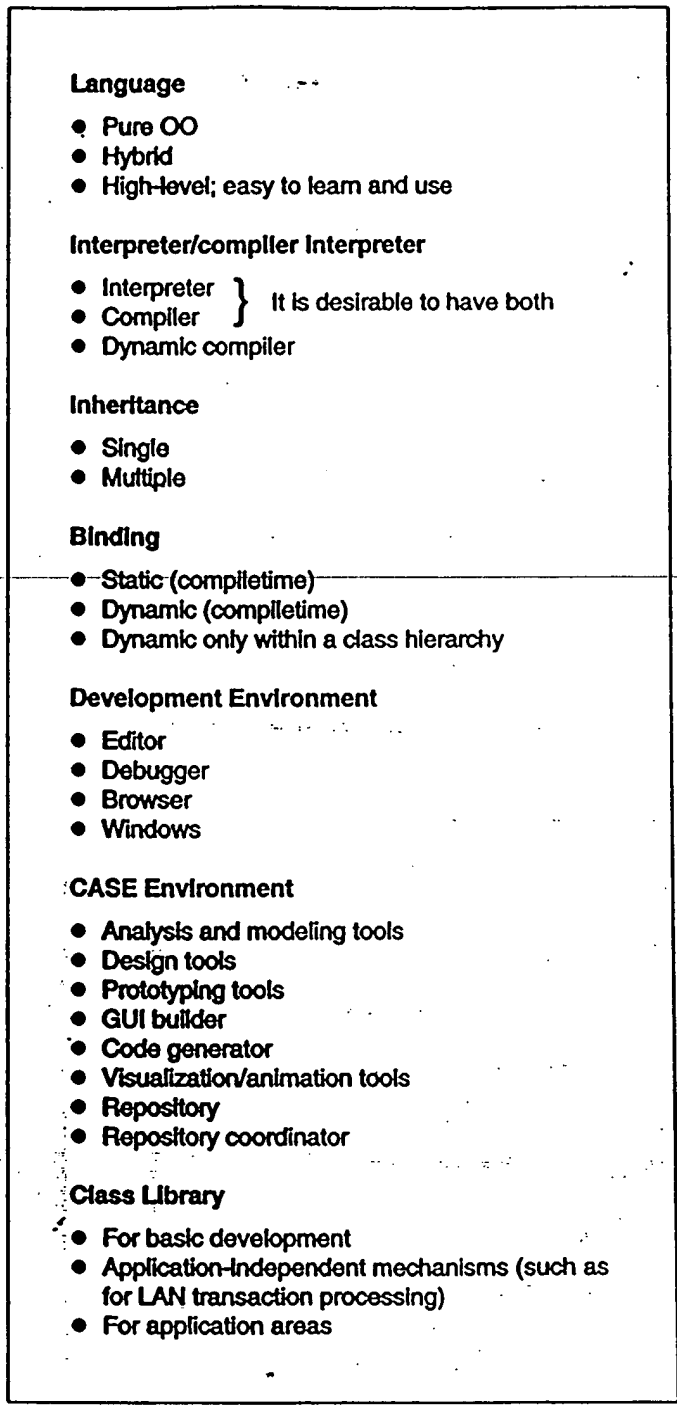


Figure 17.10 Characteristics of OO languages and development environments.

8. Khoshafian, Setrag and Razmik Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, New York, 1990.

9. Liskov, Barbara and John Guttag, *Abstraction and Specification in Program Development*, MIT Press, Cambridge, MA, 1986.

10. Nygaard, Kristen and Ole-Johan Dahl, "The Development of the Simula Language," *History of Programming Languages*, ACM SIGPLAN History of Programming Languages Conference (Los Angeles), Richard L. Wexelblat ed., Academic Press, New York, 1981, pp. 439-493.

11. Soley, Richard Mark, ed., *Object Management Architecture Guide*, Object Management Group, Document 90.17.1, November 1, 1990, Framingham, MA.

12. Taylor, David A., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, Reading, MA, 1992.

18 OO-CASE TOOLS

Object-oriented techniques and CASE technology fit naturally together. While the OO world initially emphasized OO programming, the emphasis, today, should be on repository-based development with integrated CASE tools and a powerful code generator.

OO is much more than a computer language or design technique: it is a way of thinking. CASE tools oriented to this way of thinking help the I.S. professional, as well as the business person, engineer, or end user, to visualize automation in terms of OO models and specifications.

Some non-OO CASE tools have impressive code generators which generate 100 percent of the code with no syntax errors. (There are often design errors; garbage-in-garbage-out applies to code generators) [3]. Code generators of equivalent quality ought to become the predominant way of creating OO applications, along with a CASE repository of reusable classes. With OO-CASE tools the main emphasis of system building will be modeling and design, rather than OO programming.

Several OO analysis methods already exist in the data-processing industry. However, most are based on the structure of object-oriented programming languages—rather than its fundamental principles. They begin by defining classes and superclasses and continue by specifying the data structure of the classes. Next, the operations associated with each class are identified. Since these operations must connect in some way, their interfaces or *request* structures are defined. Finally, the methods for each operation are specified. Most methodologies do not identify events, triggers, rules, and state changes (as described in Part II). These are important and should be an essential part of analysis and design methodologies.

> OO analysis methods should not be tied to OO languages and their support facilities.

Since technology for development is changing rapidly, we should analyze our systems *independently* of the programming tools used. Programming techniques should not drive the way we understand and communicate about our organizations.

A VARIETY OF TOOLS

Various tools have been created to help make I.S. development faster, cheaper, and higher quality. Some tools are simple, others are complex. Simple tools should be reserved for simple systems, where a complex tool might slow down development. Certain needs require only a report generator or a spreadsheet tool.

In the early 1980s, fourth-generation languages (4GLs) were invented [2]. Some 4GLs use nonprocedural languages that offer ways to express *what* result is required—not *how* to achieve it. SQL became a standard and provided a nonprocedural way to access relational databases. Other, more user-friendly, query languages and report-generation languages proliferated.

Most fourth-generation languages were invented before object-oriented techniques were well understood. Today, we need better end-user languages that incorporate OO techniques. Objects that have complex behavior may be shown on the screen as an icon. The end user can link many such icons together on a PC screen to build applications. This is done in some process-control software (e.g., from Gensym Inc.) and some decision-support software (e.g., from Metaphor Inc.).

Prototyping tools are important, enabling developers to build prototypes quickly and see how end users react to them. Prototyping languages gave rise to iterative development in which a prototype was successively refined.

CASE (Computer-Aided Systems Engineering) tools provided graphically oriented ways of expressing plans, models, and designs [4]. Code generators were created that could generate COBOL or other languages from high-level constructs [1].

As CASE tools became more powerful, much of the design could be synthesized rather than built from scratch. The design tools were linked to a repository containing information used in building the design. The repository stored the information involved in planning, analysis, design, and code generation. The repository contained templates and reusable constructs that could be customized for the system being built. It might contain specimen applications that could be modified. Particularly important now, the repository should contain reusable OO classes, designed to be incorporated in applications.

The tools really started to look powerful when these facilities were integrated. CASE tools for planning, for data and process modeling, and for creating designs were integrated with code generators. Prototyping capability was linked into the design tools. Nonprocedural languages, including SQL and report generators, were integrated into the CASE environment. The term I-CASE describes

integrated CASE products. Most important in I-CASE is the ability to generate code directly from the CASE design tool. An I-CASE repository contains design components in the form of reusable classes.

PURE OO AND HYBRID CASE

Like programming languages, CASE tools can be divided into three categories:

- non-OO
- pure OO
- hybrid

The CASE industry grew up in the late 1980s building non-OO tools. Because of this, OO enthusiasts have poured scorn on CASE tools [6] despite their clear success in facilitating faster development and maintenance.

As OO techniques became better understood, some CASE vendors added OO concepts to their toolsets. Their repositories made available design objects, such as screens, dialogs, reports, tables, procedures, database-access mechanisms, transaction controls, and so on. Subtyping and inheritance made it possible to use these design objects in a flexible way. The objects could be modified by users to incorporate into a particular application, as in Figs. 12.8 and 12.9.

While design objects were used and helped with prototyping and code generation, the basic CASE diagrams generally remained the same. They supported traditional structured techniques with functional decomposition, structure charts, and data-flow diagrams, rather than class hierarchies, subtyping, event diagrams, and the OO diagrams in general. The tools had a flavor of OO, but their basic core did not reflect the paradigm shift from structured techniques to OO techniques.

More recently, pure OO-CASE tools evolved. These support all or most of the techniques described in Part II of this book and they generate code. Some of these evolved on one workstation, able to support one developer rather than teams of developers.

Integrated OO-CASE tools are urgently needed. They enable development teams to share a repository, so that development can progress from intelligent enterprise models to efficient system generation.

CATEGORIES OF CASE TOOLS

CASE tools can be categorized as *I-CASE* (supporting the whole lifecycle with a single logically consistent repository) and *fragment CASE* (supporting only a portion of the lifecycle). Fragment CASE tools include front-end tools (the analysis part of the lifecycle) and back-end tools (code generation). Some CASE tools

support information engineering (see Chapter 16) and help their users to build a model of the enterprise and analyze business areas. We thus have IE-CASE (for complete information engineering), I-CASE (integrated CASE that is not necessarily information engineering), and fragment CASE. Any of these can be non-OO, traditional but with an OO flavor, pure OO, or can support *both* traditional techniques and OO techniques. This categorization is shown in Fig. 18.1.

		Non-OO	Traditional with an OO flavor	Both OO and non-OO	Pure OO
Fragmented CASE	Front-end tools				
	Back-end tools				
Integrated CASE (but not IE)					
IE CASE (Complete Information Engineering)					

	Non-OO	Traditional with an OO flavor	Both OO and non-OO	Pure OO
Diagrams for traditional structured techniques	3	3	3	
Diagrams for OO techniques			3	3
Design objects with inheritance		3	3	3

Figure 18.1 Categories of CASE tools

INSIGHTFUL MODELS

Figure 18.2 repeats Fig. 5.1 and shows how we model and design systems. The capabilities in this figure should be those of a CASE toolset. The methodology supported by the toolset should ensure that the modeling leads directly to design as automatically as possible and that code is generated immediately from the design.

The model of reality should be as meaningful as possible to end users. In corporate I.S. we build a model of the enterprise. This should reflect the way the business people want to run the enterprise and facilitate discussion of how the enterprise procedures should be changed.

The early CASE tools modeled the enterprise with functional decomposition, entity-relationship diagrams, and matrices mapping entity types against functions. This modeling did not reflect how the enterprise operated and was not very meaningful to business people. OO modeling is more effective, because we identify events and the operations triggered by events. Event diagrams show streams of operations and allow us to reorganize the stream and redesign the busi-

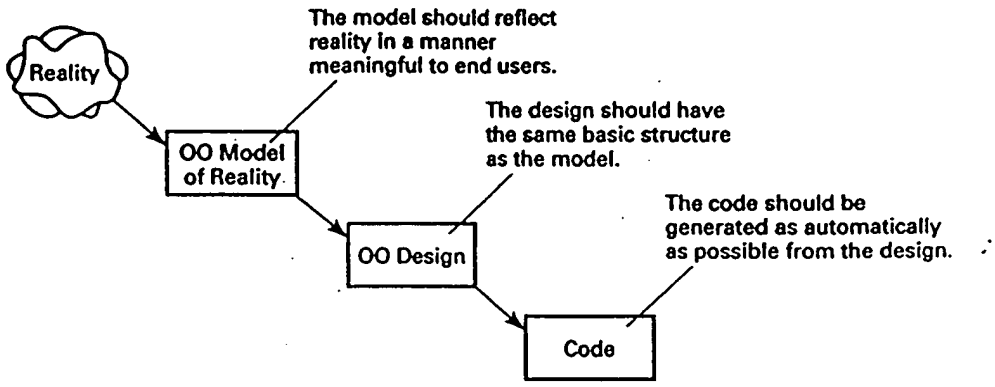


Figure 18.2 CASE tools should enable us to model reality in a manner as insightful as possible to end users and to translate those models as automatically as possible into the systems the users need.

ness process. The value added by each operation may be analyzed and operations of low value eliminated. Various business rules state how the business should react to events, such as what to do about late payers, what factors have priority in shop-floor scheduling, and so on. These rules relate to business objects. We identify the business objects, operations that change their states, and the rules that should govern these operations.

> The challenge of CASE tools today is modeling reality in a manner as insightful as possible to end users and translating the models as automatically as possible into the required systems. OO techniques help us to do that.

DESIGN SYNTHESIS AND CODE GENERATION

The most powerful I-CASE tools allow as much design as possible to be *synthesized* from high-level constructs in the repository. The designer assembles the design and creates its detailed logic. Some design may be generated from high-level behavioral and structural statements such as state-transition diagrams, rules, decision trees, event diagrams, and object schemas [4].

The design feeds a code generator that creates 100 percent of the code with zero programming errors. The generated code should never have an ABEND. The testing process is geared more for catching the *design* errors than catching detailed coding bugs.

The combination of *design synthesis* and *code generation*, shown in Fig. 18.3, enables high-quality applications to be built quickly. A world of difference exists between the modern development lifecycle using the tools in Fig. 18.3 and

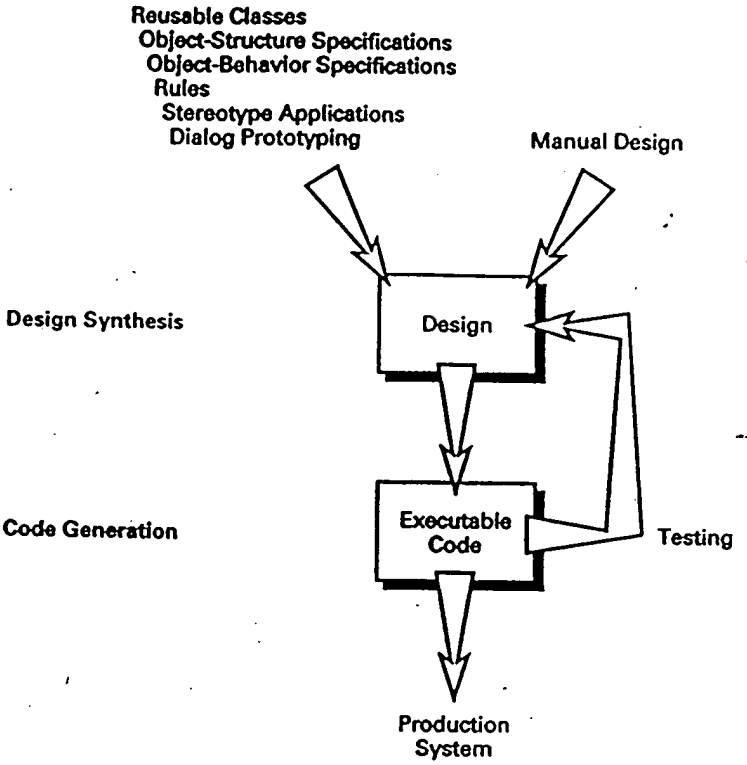


Figure 18.3 Design synthesis and code generation.

the traditional building of systems with plastic templates and line-by-line COBOL coding.

As far as possible, the design should be synthesized from

- Object-structure diagrams and specifications
- Object-behavior diagrams and specifications
- Rules
- Application-independent classes
- Classes for specific applications
- Entire applications, designed for customization
- Designs that can be customized
- Declarative tables (e.g., decision tables, event-condition-action tables)
- A report generator
- A screen painter
- A dialog prototyping tool (GUI)

CODE GENERATORS

Pure OO-CASE tools need not necessarily use OO programming languages. They can generate code in languages such as COBOL or C—both of which are efficient and portable and have highly efficient optimizing compilers.

A problem with COBOL and C compilers is that they do not provide dynamic (runtime) binding which has the advantages described in the previous chapter. Some OO-CASE tools generate C++ which provides dynamic binding as a requestable option but only for classes that belong to the same superclass.

A principle of integrated CASE is that debugging does not involve changing instructions at the code level, but rather changing the design that drives the interpreter or code generator. The interpreter or generator creates code with no syntax errors—this code can be tested like a black box and the source changed when necessary. This is particularly important if the tool generates a traditional language, such as COBOL or C, at the code level.

We will refer to this immediate instantiation as *instant CASE*. It helps the designer to be highly creative, observing his creation as he builds it, like a sculptor, and modifying it as it evolves. Developers find instant CASE far more satisfying than traditional CASE tools which require a long wait for code generation and compilation. With the latter, it takes too long to modify and improve what is being built.

INSTANT CASE

Building systems with an interpreter is highly desirable so that the developer can immediately run what he creates and repeatedly modify and evolve it. As with programming languages, a CASE toolset should have both an interpreter and an optimizing compiler.

With OO tools, as soon as an object type is described to the tool, it can create tables that allow instances of the object to be created. A description of its data can be built and tables of instances filled in. The designer can create subtypes, composed-of hierarchies, and object-relationship diagrams and immediately fill in details of instances. Methods can be created and the behavior of the objects observed, tested, and adjusted.

As soon as the designer describes object types to the tool, the objects *are there* and can be experimented with. This resembles a spreadsheet tool in which the designer can add columns and rows, specify computations, and immediately add values and run the spreadsheet, generating and modifying charts. This ability of spreadsheet tools to create *instant* results should be emulated with OO-CASE. As soon as object types and methods are created, they can be observed in action. The designer can build up structures far more complex than spreadsheets in an interactive manner, can experiment with them, and test them, as a spreadsheet designer does.

PRECISION IN DIAGRAMMING

An article in *Fortune* [5] attempts to describe why software is so difficult to create:

> Software is "pure thought stuff," so conceptual that designers cannot draw explicit, detailed diagrams and schematics—as creators of electronic circuits can—to guide programmers in their work. Consequently, routine communication among programmers, managers and the ordinary people who use software is a chore in itself.

This popular wisdom is *wrong* today. With the I-CASE tools, explicit, detailed diagrams and schematics are drawn, analogous to those used by electronic circuit designers, and code is generated from them. Much testing can be done at the diagram level. These diagrams are very effective for routine communications among programmers, analysts, managers, and end users. Just as other engineering disciplines have precision represented in formal diagrams, so computer system developers also need formality and precision—in this case, enforced by the computer.

Given appropriate diagramming techniques, describing complex activities and procedures is easier through diagrams than text. A picture can be much better than a thousand words, because it is concise, precise, and clear. Computerized diagrams do not allow the sloppiness and woolly thinking common in textual specifications. Engineers of different types use formal diagrams that are precise in meaning—mechanical drawings, architectural drawings, circuit diagrams, microelectronics designs, and so on. The diagrams become the documentation for systems (along with the additional information collected in the repository when the diagrams are drawn).

A good CASE tool employs diagram types that are precise and can be checked by computer. Large, complex diagrams can be handled by means of zooming, nesting, windowing, and other computer techniques. The computer quickly catches errors and inconsistencies even in very large sets of diagrams. Business, government, and the military need highly complex, integrated computer applications. The size and complexity of these applications require accurate diagramming using computers.

The *meaning* represented by the diagram is more valuable than its graphic image. A good CASE tool stores that meaning in a computer-processable form. The tool helps build up a design, a data model, or some other deliverable segment of the development process, so that it can be validated and used in a subsequent development stage.

> The evolution of OO-CASE technology represents an evolution of application development from being a *craft* to being an *engineering discipline.*

DIAGRAMS FOR OO DEVELOPMENT

For OO analysis and development, CASE tools should enable their users to build the diagram types listed in Box 18.1. Many of these diagrams are the same as those for non-OO development:

- Data structure diagrams (showing records, their fields, keys, and interrelations)
- Action diagrams (showing the structure of procedural code)
- Declarative tables (e.g., decision tables or event-condition-action tables)
- Tools for designing the graphic user interface
- State-change diagrams (showing the state changes of an object)

Some of the diagrams are essentially the same as those used in conventional data-centered development but with an OO flavor:

BOX 18.1

CASE tools for OO analysis and design should enable their users to build the following types of diagrams and generate code from them:

For Object Structures:

- Data-structure diagrams
- Object generalization diagrams
- Object-relationship diagrams
- Object composed-of diagrams

For Object Behavior:

- Class-communication diagrams
- Representations of methods
  - — Action diagrams
  - — Declarative tables
- State-change diagrams
- Event diagrams
- Rules, linked to event diagrams
- Tools for designing the graphical user interface

17 OBJECT-ORIENTED PROGRAMMING LANGUAGES

THE GENESIS OF OO TECHNOLOGY

The genesis of the technology now called *object-oriented* dates back to the early 1960s. It arose from the need to describe and simulate a variety of phenomena such as nerve networks, communications systems, traffic flow, production systems, administrative systems, and even social systems. In the spring of 1961, Kristen Nygaard originated the ideas for a language that would serve the dual purpose of system description and simulation programming. Together with Ole-Johan Dahl, Nygaard developed the simulation language now known as Simula I. The first Simula-based application package was implemented for the simulation of logic circuits. However, operations research applications were the most popular usage. For example, in 1965 a large and complex job shop simulation was programmed in less than four weeks—with an execution efficiency at least four times higher than that of available technology [4,10].

Simula was intended to be a system description and simulation programming language. However, its users quickly discovered that Simula also provided new and powerful facilities when used for purposes *other than* simulation, such as prototyping and application development. In September 1965, the possibilities of a "new, improved Simula" as a general purpose language were being planned. By December 1966, the necessary foundation for the new, general programming language, called Simula 67, was defined.

SMALLTALK

In the late 1960s, another development of OO technology was under way, guided by research at the University of Utah and by the central ideas of Simula. Alan Kay envisioned that by the 1980s

> [B]oth adults and children will be able to have as a personal possession a computer about the size of a notebook with the power to handle virtually all their information-related needs. . . . Ideally the personal computer will be designed in such a way that all ages and walks of life can mold and channel its power to their own needs [7].

Early in the 1970s, Alan Kay went to Xerox and formed the Learning Research Group. Xerox was responsible for producing the interim model for the personal computer, called Dynabook. The group was engaged to produce the software, called Smalltalk. They quickly realized that one of the major design problems involved expressive communication—particularly when children were seriously considered as users. For this reason, the group invited some 250 children (aged six to fifteen) and 50 adults to try versions of Smalltalk and suggest ways of improving it. In order to test the usability of Smalltalk, they started with simple situations that embodied a *concept* and gradually increased the complexity of the examples. A major goal of Smalltalk was providing a single name (or symbol) for a complex collection of ideas. Later, these ideas could be invoked and manipulated through the name. They found that children by the age of six were able to do this.

While the Dynabook project did not realize its goal, Smalltalk evolved into an important OO language. Alan Kay foresaw the need to characterize and communicate application *concepts* in developing computer programs. Smalltalk provides the means to write programs in a style that brings our concepts to life. The term *object-oriented* originated during the development of Smalltalk [1, 6].

THE EVOLUTION FROM UNTYPED TO TYPED LANGUAGES

The kinds, or *types*, of data on which a program can operate need to be organized. Initially, only one data type described the universe of bit strings in a computer memory—the data type Word. Words are bit strings of fixed size that can be used as units of information.

The need for data types arises whenever data must be categorized for a particular usage. As early as 1954, FORTRAN distinguished between Integer and Floating-point types of data. Later, Algol 60 incorporated data types for Integer, Real, and Boolean. Still later, languages included additional data types, such as the Character, String, Bit, Byte, Array, Pointer, Record, File, and Procedure.

A *data type* describes a certain kind of data—its representation and the set of valid operations that access and maintain that data. In this way, each data type is a known commodity, protected from unintended use. For example, the data type Character describes the kind of data that is displayable by a program. Furthermore, a set of operations is provided for creating, destroying, examining, and manipulating Character data. Since arithmetic operations such as add and subtract are not defined for Character data, computational requests are not permitted [9].

USER-DEFINED TYPES (UDTs)

Prior to the early 1970s, a programmer could reference only those data types built into a programming language compiler. As a result, even often-used types such as Month, Date, Time, Coefficient, Tree, and Stack were not explicitly accessible. These ideas had to be implicitly embedded somewhere in the programmer's code. An additional limiting characteristic of built-in types was their definition by the way in which the information was physically stored. They had little useful relationship to the real-world objects that the application was trying to implement.

Eventually, the computer industry felt pressured to provide programmers with a facility for expressing their own typing needs. The first languages to offer *user-defined types* (UDTs) were Pascal and Algol 68. In Pascal, for example, a programmer could write

```
type month = (January, February, March, April, May, June, July, August,
   September, October, November, December);
```

This expression defines the UDT Month as being the set of twelve literals. The developer could define relational operations to compare two given Month variables for less than, equal to, and so on. Other operations could include computing the preceding or succeeding month when supplied with a Month variable.

The *types* of Pascal and the *nodes* of Algol 68 were an important step forward. They permitted the programmer to go from manufacturer-imposed types to user-imposed types. UDTs raised the expressive power of programming languages. More importantly, they encouraged systems developers to translate the real-world types of the system application into coded data types.

ABSTRACT DATA TYPES (ADTs)

The *abstract data type* (ADT) extends the notion of the *user-defined type* (UDT) by adding encapsulation. The ADT contains the representation and the operations of a data type. The *encapsulation* feature of the ADT not only hides the data type's implementation but provides a protective wall that shields its objects from improper use. All interfacing occurs through named *operations* defined within the ADT. The operations, then, provide a well-defined means for accessing the objects of a data type. In short, ADTs give objects a *public* interface through its permitted operations. However, the representations and executable code, or *method*, for each operation are *private*.

The ADT facility first appeared in Simula 67. Its implementation is called a *class*. Modula refers to its ADT implementation as a *module*, while Ada uses the word *package*. In all cases, the ADT provides a way for the systems developer to identify real-world data types and package them in a more convenient and compact form. In this way, ADTs can be defined for things such as Dates, Screen Panels, Customer Orders, and Part Requisitions. Once defined, the developer can address the ADTs directly in future operations.

Workflow Template

Process Template

Using the WFT Development Environment

TEMPLATE SOFTWARE

License Information

This document describes Release 8.0 of Workflow Template (WFT). This document is subject to change without notice, and Template Software assumes no responsibility for any errors that may appear in this document. The WFT software described in this document is furnished under a license agreement. It is unlawful to copy WFT on any medium for any purpose other than the licensee's use.

Trademark Acknowledgments

SNAP is a registered trademark of Template Software, Inc. Workflow Template is a trademark of Template Software, Inc.

Borland C++ is a trademark of Borland International, Inc. Alpha AXP, AXP, DEC, DECnet, DECstation, DECwindows, Open VMS AXP, VAX, VAXstation, VMS, and ULTRIX are trademarks of Digital Equipment Corporation. HP and HP-UX are registered trademarks of Hewlett-Packard Company. INFORMIX is a registered trademark of Informix Software, Inc. InterBase is a trademark of Interbase Software Corporation. DATABASE2, DB2, IBM, and OS/2 are registered trademarks of International Business Machine Corporation. AIX, C Set++, and RISC System/6000 are trademarks of International Business Machine Corporation. IXI Motif is a trademark of IXI Limited. X Window System is a trademark of Massachusetts Institute of Technology. Microsoft, MS, and MS-DOS are registered trademarks of Microsoft Corporation. Visual C++, Windows, Windows NT, WIN32, and WIN32s are trademarks of Microsoft Corporation. Motif and OSF/1 are registered trademarks of Open Software Foundation, Inc. ORACLE, Oracle7, and SQL*Plus are registered trademarks of Oracle Corporation. Pro*C is a trademark of Oracle Corporation. OSx and Pyramid are registered trademarks of Pyramid Technology Corporation. DC is a trademark of Pyramid Technology Corporation. Solaris, Sun, SunOS, and SPARC are trademarks of Sun Microsystems, Inc. SYBASE is a registered trademark of Sybase, Inc. DB-Library, Open Client, SQL Server, and SQL Toolset are trademarks of Sybase, Inc. MultiNet is a registered trademark of TGV, Inc. SCO is a trademark of The Santa Cruz Operation. Unisys is a trademark of Unisys Corporation. UNIX are registered trademarks of X/Open Company Limited.

Copyright Notice

If you have any comments concerning this document or software, please forward them to:

Template Software, Inc.
45365 Vintage Park Plaza, Suite 100
Dulles, VA 20166
Internet address: support@template.com

CHAPTER

USING THE DEPLOYMENT EDITOR

About this chapter

This chapter describes the tools, menus, and other mechanisms provided by the Deployment Editor for creating and editing the deployment configuration for your workflow system.

Contents

Introduction

The **Deployment Editor** enables you to map the business process applications and servers in your WFT workflow system to the physical architecture of your computer network. Business process applications are mapped to business process nodes, and servers are created as server nodes, or servers. This prepares your WFT workflow system for deployment in the physical configuration of your network. The Deployment Editor then enables you to actually deploy your workflow system. Through the Deployment Editor, you can specify the communication protocols and addresses used for business process nodes and servers. You also specify the services that each server provides. You can also specify backups for servers. The Deployment Editor also enables you to develop a list of end users and their passwords for each business process node in the deployed system.

See *Chapter 7, Deploying a WFT Workflow System*, in *Developing a WFT Workflow System* for more information about the concepts and process of deploying a workflow system.

In this chapter, you will find how to:

- Create business process nodes and servers in your deployed workflow system.
- Specify communication protocols and addresses for business process nodes and servers.
- Map the communication links between business process nodes and servers and the services provided over these links.
- Specify backup information for servers.
- Develop lists of end users for the business process nodes in your deployed workflow system.

Prerequisites

Before you use the Deployment Editor, you must create applications for your WFT workflow system.

Important terms

The terms defined in Table 8-1 include some of the basic terms you need to understand to use the Deployment Editor. More detailed definitions and definitions of additional terms are included in *Appendix A, The Workflow Template Glossary,* in *Developing a WFT Workflow System.*

Table 8-1. Important terms

Term	Definition
Application	A set of related tasks. An application can be composed of from zero to many tasks.
Build	To compile the source files for a software development effort into object files that can be executed. In the WFT, you can build at the task and application levels. Building a business process node or server is the same as building an application that runs in the business process node or server.
Business process application	An application that includes related tasks that are generally analogous to the job functions of a particular kind of employee.
Business process node	A named instance of a business process application in a WFT workflow system.
Communication link	The communication pathway between a business process node and a server in a WFT workflow system. A communication link is defined by the business process node and server it connects and the communication protocol it uses.
Deployment	The process of defining the run-time configuration of a WFT workflow system by designating the relationships among the logical and physical elements of the system; the mapping of servers and business process nodes to the physical architecture of the computer network.
Persistent storage	A storage area, associated with an individual node, that contains both the stored copies of the messages sent by the tasks within the node's application and the copies of any unprocessed work items for each task within the node's application. This repository for message and work item traffic is vital to the reconstruction of processing status and data integrity when restarting or replacing a failed node.
Server	A node that performs specialized processing for business process applications and/or other servers, rather than performing business-process-related processing. Business process tasks cannot run in servers.

Accessing the Deployment Editor

To access the Deployment Editor, use either of the following methods:

Method 1:

➤ Click the Deployment Editor button  in the WFT Development Environment main window.

Method 2:

➤ Choose **Deployment Editor** from the *Editors* menu in the WFT Development Environment main window.

The Deployment Editor appears, as shown in Figure 8-1.

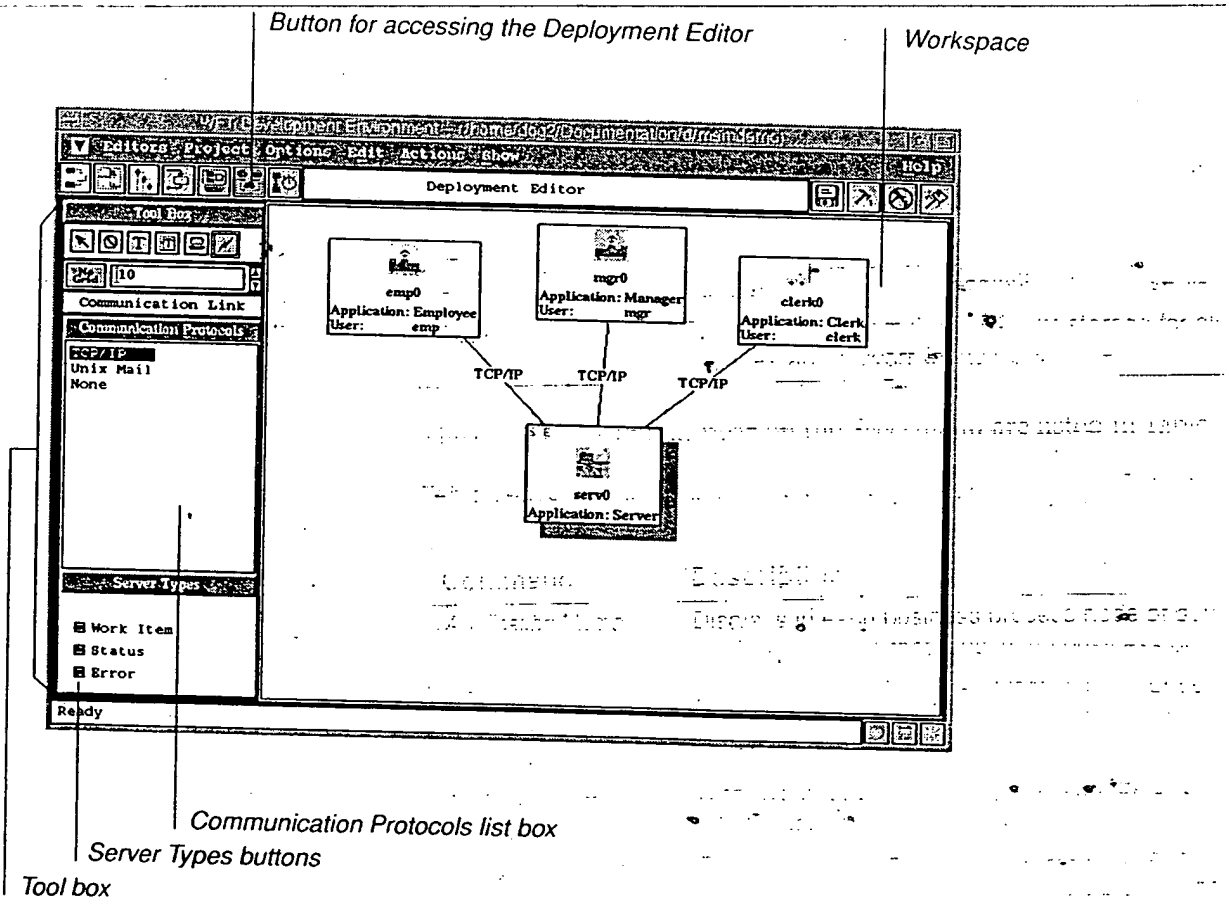


Figure 8-1. The Deployment Editor

The Deployment Editor provides menus and tools for defining nodes and communication links.

About the Deployment Editor menus

The Deployment Editor provides *Edit*, *Actions*, and *Show* menus. The ▼, *Editors*, *Project*, *Options*, and *Help* menus are common to all WFT Development Environment editors and are described in *About the status line* on page 2-15 in *Chapter 2, Introduction to the WFT Development Environment*.

The commands that appear on the *Edit* menu are listed in Table 8-2.

Table 8-2. Commands provided on the Deployment Editor *Edit* menu

Command	Description
Set Workspace Size	Enables you to manually change the size of the workspace.
Auto Size Workspace	Automatically resizes the workspace to the smallest possible size that can contain all of the workspace elements.
Storage Backup Frequency	Enables you to specify when a copy of persistent storage for servers will be spooled. The copy can be copied to a secure storage device so that the system can be recovered after a catastrophic failure, such as a hard disk crash.

The commands that appear on the *Actions* menu are listed in Table 8-3.

Table 8-3. Commands provided on the Deployment Editor *Actions* menu

Command	Description
Run All	Runs all the business process nodes and servers in the workspace.
Clean Storage	Removes data from the persistent storage for all the business process nodes and servers in the workspace.

The commands that appear on the *Show* menu are listed in Table 8-4.

Table 8-4. Commands provided on the Deployment Editor *Show* menu

Command	Description
Application Name	Displays in each business process node or server symbol the name of the application that runs in the business process node or server.
Host Name	Displays in each business process node or server symbol the name of the host computer on which the business process node or server will run.
Storage Name	Displays in each business process node or server symbol the name of the business process node or server's persistent storage area.
Users	Displays in each business process node or server symbol the list of users that are defined for that business process node or server.
Bitmap	Displays in each business process node or server symbol the bitmap selected for that business process node or server.

About the Deployment Editor tool box

The Deployment Editor tools, buttons, text entry line, list box, and pop-up menus are described in the following sections. Figure 8-2 shows two examples of the Deployment Editor tool box.

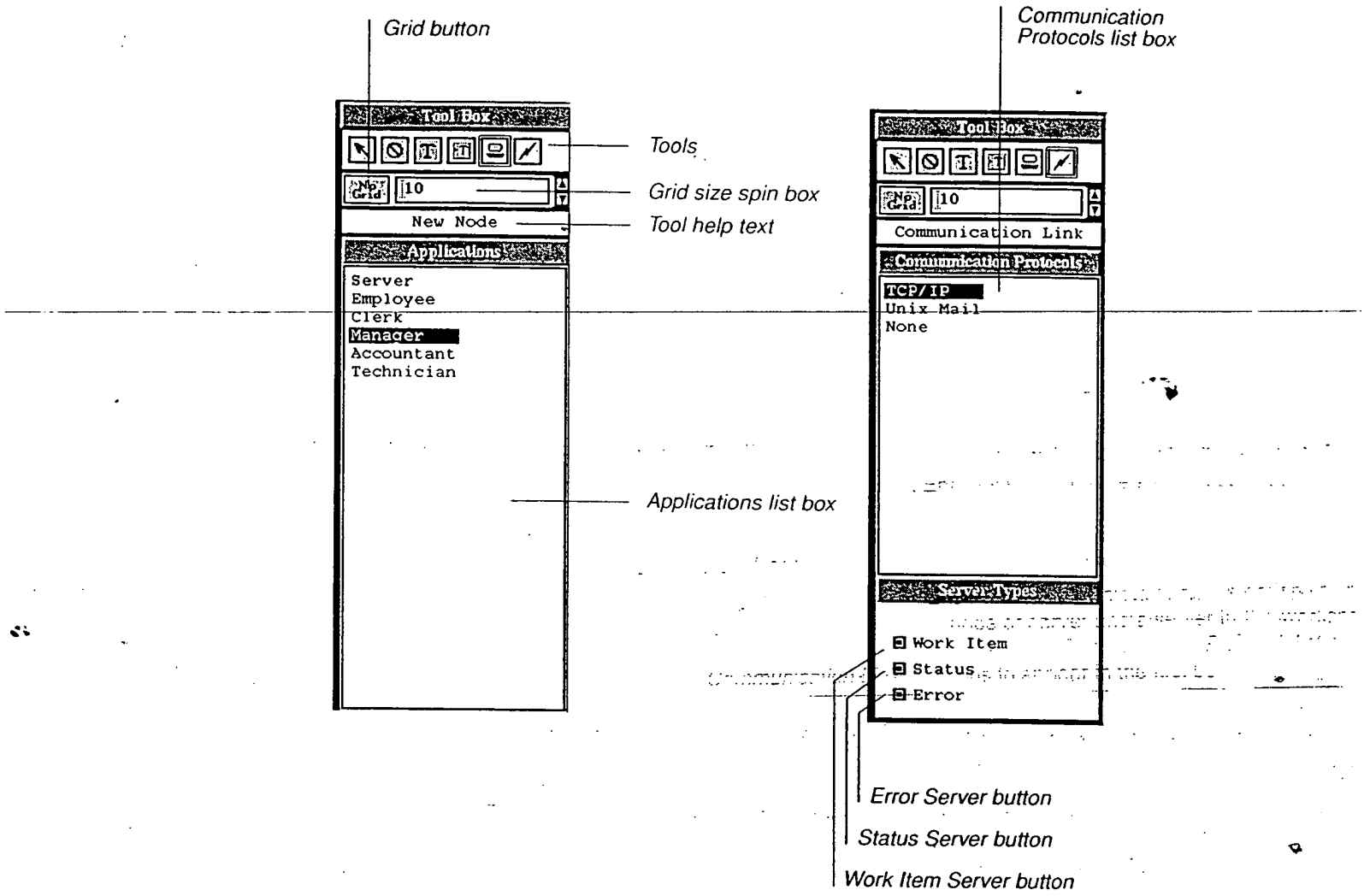


Figure 8-2. Example Deployment Editor tool boxes

Deployment Editor tools

The tools that appear in the tool box are listed in Table 8-5. In the tool box, the name of the currently selected tool is displayed in the tool help text area.

Table 8-5. Tools provided in the Deployment Editor tool box

Tool	Description
 *Select*	Enables you to select items in the workspace.
 *Remove*	Removes items from the workspace.
 *Edit Text In Place*	Enables you to edit communication link labels and some text information displayed in the business process node or server symbols where it appears in the workspace.
 *Edit Text*	Enables you to edit, in an edit window, communication link labels and some text information displayed in the business process node or server symbols in the workspace.
 *New Node*	Enables you to create a business process node or server in the workspace. Selecting this tool causes the *Applications* list box to appear in the tool box.
 *Communication Link*	Enables you to create a communication link between a business process node or server and a server in the workspace. Selecting this tool causes the *Communication Protocols* list box and the *Server Types* buttons to appear in the tool box.

Deployment Editor tool box buttons

The grid button enables you to use the grid to help you place and align elements in the workspace. The grid is a ruled background on which you can snap elements to a particular location. The grid button options are listed in Table 8-6.

Table 8-6. Deployment Editor tool box grid button options

Grid options	Description
	*No Grid* – Turns the grid off, whether it is visible or not.
	*Grid On* – Turns the grid on and makes it visible. When the grid is on and visible, the grid button displays a graphic representation of a grid.
	*Grid Invisible* – Turns the grid on, but makes it invisible. When the grid is on, but invisible, the grid button displays a gray, shadow-like graphic representation of a grid.

Deployment Editor tool box spin box

The spin box that appears in the tool box is listed in Table 8-7.

Table 8-7. Spin box provided in the Deployment Editor tool box

Spin box	Description
Grid Size	Enables you to set the size of the grid squares, in pixels. Using the up and down arrows, you can increase and decrease the grid size within the range of values from 2 to 400, inclusive.

Deployment Editor *Applications* list box

The *Applications* list box displays the names of all of the applications currently defined for your WFT workflow system. You deploy these applications as business process nodes or servers in your workflow system. This list box is only available when you select the *New Node* tool.

Deployment Editor *Communication Protocols* list box

The *Communication Protocols* list box displays the names of all of the communication protocols currently supported by your WFT workflow system, such as TCP/IP and UNIX mail. You select one of the supported protocols when you define a communication link. This list box is only available when you select the *Communication Link* tool.

Deployment Editor *Server Types* buttons

The *Server Types* buttons enable you to select the specific types of services that servers provide. You select one or more of the types of services for each communication link you create. These buttons are only available when you select the *Communication Link* tool. The *Server Types* buttons are listed in Table 8-8.

Table 8-8. *Server Types* buttons in the Deployment Editor

Button	Description
Work Item	Specifies that the server with which a given business process node communicates via a given link provides work item services to that node.
Status	Specifies that the server with which a given business process node communicates via a given link provides work item services to that node.
Error	Specifies that the server with which a given business process node or other server communicates via a given link provides error services to that business process node or server.

About the Deployment Editor workspace

The Deployment Editor workspace, which is the large portion of the display to the right of the tool box, is the area in which you configure your workflow system's deployment. You draw the various elements of your workflow system in this workspace, as though it were a large piece of drawing paper. The tool box provides controls for the grid in the workspace to help you align the representations of your workflow system elements.

As you can do with many popular drawing programs, you can view this workspace display as a window onto a larger virtual workspace. The Deployment Editor gives you a very large virtual workspace, one that you probably will not need to resize. Because the virtual workspace typically is larger than the display area in the window, scroll bars appear along the right side and the bottom of the workspace.

If you choose the *Print* command from the ▼ menu, the entire virtual workspace is printed, not just the currently displayed window onto the larger virtual workspace.

Figure 8-3 shows an example Deployment Editor workspace.

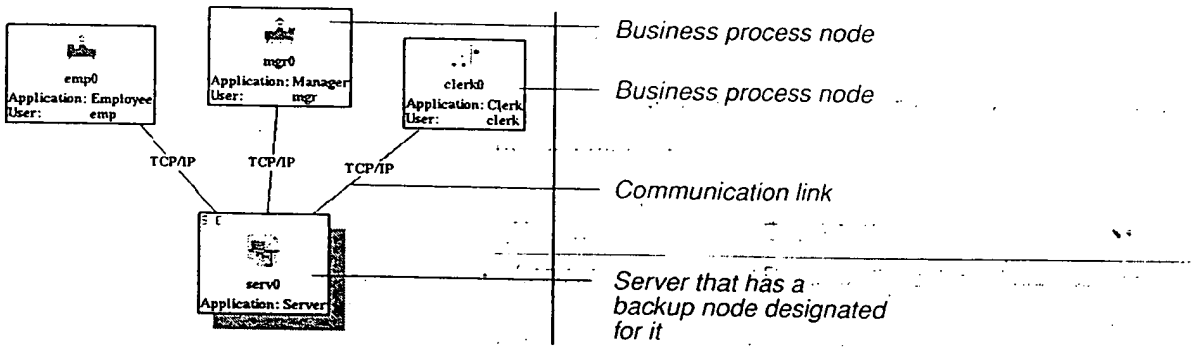


Figure 8-3. Example Deployment Editor workspace

The symbol that represents a server can contain two graphic indicators. An S in the upper left corner of the server's symbol indicates that the server provides status services to some or all of the business process nodes and servers that are connected to it. Similarly, an E in the upper left corner of the server's symbol indicates that the server provides error services to some or all of the business process nodes and servers that are connected to it.

About the Deployment Editor pop-up menus

The Deployment Editor provides pop-up menus for the following workspace display elements:

- Business process nodes and servers
- Communication links
- Communication link line segment circles

Pop-up menus appear when you click the appropriate workspace element using a non-left mouse button. The pop-up menus do not always contain all of the commands. The commands that are included at any given instant depend on the situation; commands that cannot be used in a given situation typically are not included in the pop-up menu at that time. The entire list of possible commands for the pop-up menus is described in the following sections.

Pop-up menu for business process nodes and servers

The commands that appear on the pop-up menu for business process nodes and servers are listed in Table 8-9.

Table 8-9. Commands provided on the Deployment Editor pop-up menu for business process nodes and servers

Command	Description
Remove	Removes the selected business process node or server from the workspace.
Edit Text In Place	Enables you to edit some text information where it appears in the symbol for a selected business process node or server. This command is only available when you access the pop-up menu from a text item that you can edit in the symbol for a business process node or server.
Edit Text	Enables you to edit in an edit window some text information that is displayed in the symbol for a selected business process node or server. This command is only available when you access the pop-up menu from a text item that you can edit in the symbol for a business process node or server.
Set Host	Enables you to specify the name of the host on which the selected business process node or server will run.
Show Server Paths	Displays the node paths to the servers that provide services to a selected business process node or server.
Edit Server Backup	Enables you to edit the backup information for a selected server. This command is only available when you access the pop-up menu from a server.
Remove Server Backup	Removes the backup information for a selected server. This command is only available when you access the pop-up menu from a server for which backup information has been specified.

Table 8-9. Commands provided on the Deployment Editor pop-up menu for business process nodes and servers (cont)

Command	Description
Bitmap	Enables you to set the display of a bitmap within the symbol for a selected business process node or server. This command provides the following additional commands: • *Set Bitmap File* – Enables you to select a bitmap to be displayed within the symbol for a selected business process node or server, in place of the default bitmap. • *Don't Use Bitmap File* – Removes any bitmap from the symbol for a selected business process node or server.
Storage Backup Directory	Enables you to specify the directory in which WFT places a copy of the selected server's data.
Storage Settings	Enables you to specify how to verify that data have been placed in persistent storage and how many items will be put into each persistent storage file.
Deployment Data	Enables you to edit the definition and specification information for a selected business process node or server. This command provides the following additional commands: • *Node Name* – Enables you to edit the name of a selected business process node or server. • *Node User(s)* – Enables you to edit the list of user names and associated passwords for a selected business process node. • *Node Arguments* – Enables you to edit the command line arguments to be passed to a selected business process node or server at run time. • *Node Storage Name* – Enables you to edit the name of the persistent storage area for a selected business process node or server.
Deploy	Enables you to prepare and deploy a selected business process node or server. This command provides the following additional commands: • *Build* – Builds the executable files for the application that will be deployed in a selected business process node. For a server, builds the executable file for the server. • *Run* – Runs the application for a selected business process node, if possible, according to the information specified for that business process node. Runs a selected server, if possible. • *Debug* – Runs the SNAP Debugger for the node's application. • *Clean Storage* – Removes the persistent storage for a selected business process node or server.

Pop-up menu for communication links

The commands that appear on the pop-up menu for communication links are listed in Table 8-10.

Table 8-10. Commands provided on the Deployment Editor pop-up menu for communication links

Command	Description
Remove	Removes a selected communication link from the workspace.
Edit Text In Place	Enables you to edit the label of a selected communication link where it appears. This command is only available when you access the pop-up menu from a communication link label.
Edit Text	Enables you to edit the label of a selected communication link in an edit window. This command is only available when you access the pop-up menu from a communication link label.
Split	Splits a selected communication link into segments so that you can rearrange the link's graphic representation in the workspace.
Show Label	Displays the label associated with a selected communication link.
Set Protocol Values	Enables you to specify the communication protocol-specific values to be used by a selected communication link when it is used to communicate with a server.
Set Backup Protocol Values	Enables you to specify the communication protocol-specific values to be used by a selected communication link when it is used to communicate with a server's backup. This command is only available when you access the pop-up menu from a communication link that is connected to a server for which a backup has been specified.
Set Protocol	Enables you to specify the communication protocol to be used by a selected communication link. This command provides the following additional commands: • *TCP/IP* – Specifies that a selected communication link uses the TCP/IP communication protocol. • *Unix Mail* – Specifies that a selected communication link uses the UNIX mail communication protocol. • *None* – Specifies that the communication protocol for a selected communication link is currently undefined.

Pop-up menu for communication link line segment circles

The command that appears on the pop-up menu for communication link line segment circles is listed in Table 8-11.

Table 8-11. Command provided on the Deployment Editor pop-up menu for communication link line segment circles

Command	Description
Remove	Removes a selected communication link line segment circle from the link and joins the two adjacent link line segments into a single communication link line segment.

Working with business process nodes, servers, and communication links

This section provides step-by-step information about using the Deployment Editor to create either business process nodes where business process applications run or servers that provide services to other nodes, to create communication links between business process nodes and servers, and to specify deployment data.

Creating business process nodes and servers

To create a business process node or server, use either of the following methods:

Method 1:

1. Select the **New Node** tool.

   The *Applications* list box appears in the tool box.

2. Select from the *Applications* list box the business process application or type of server that you want to deploy in the business process node or server you are creating.

3. Click the point in the workspace where you want the business process node or server to appear.

Method 2:

1. Select the **New Node** tool.

   The *Applications* list box appears in the tool box.

2. Drag the business process application or server that you want to deploy from the *Applications* list box and drop it into the workspace where you want the business process node or server to appear.

A new business process node or server symbol that contains the name of the business process node or server appears in the workspace. The WFT names a business process node or server with its application name suffixed with a number. The suffix number represents the number of the instance of the particular business process node or server. The first instance is suffixed with *0* and the second with *1* and so on. For example, the first instance of a business process node where an employee (*emp*) application runs is named *emp0*.

Creating communication links and specifying services

You can draw communication links only from a business process node or server to a server, but not from a server to a business process node. The business process node or server from which you draw the link typically originates communication over the link. The server to which you draw the link typically communicates only in response to a business process node or another server that initiates communication.

To create a communication link, perform the following steps:

1. Select the **Communication Link** tool.

   The *Communication Protocols* list box and the *Server Types* buttons appear in the tool box. By default, all three *Server Types* buttons are selected, in anticipation that you want the server to provide all three types of services to the business process node or server that is connected to it by the communication link you are creating.

2. Select from the *Communication Protocols* list box the type of communication protocol that you want to be used on the communication link you are creating.

3. Click *Server Types* buttons, as appropriate, to specify the types of services the server provides over the communication link you are creating.

   Selecting and deselecting these buttons determines the types of services the server provides to the business process node or server that is connected to it by the communication link. A server can provide any or all of these services. A business process node or server can be connected to as many as three servers, each of which provides a different one of the three types of services.

4. Click and hold the left mouse button on the symbol for the business process node or server from which you want the communication link to originate.

5. Drag the cursor to a position on the symbol for the server with which you want the first business process node or server to communicate and release the mouse button.

   If the business process node or server at either end of the communication link requires a new set of communication protocol values (as determined by the WFT), a dialog box prompts you to enter the communication protocol values for the communication link you created. You can specify whether the protocol values you enter apply to the server or to the business process node that is connected to the server. Whenever possible, default values are presented for the various data that you can enter.

6. Enter the data for the protocol values or accept the default values and press **OK**.

A new communication link symbol appears in the workspace. This symbol connects the originating business process node or server with its server. While the communication link can connect one server to another server, the originating server receives services in this case.

Removing workspace elements

To remove workspace elements, use either of the following methods:

Method 1:

1. Select the **Remove** tool.
2. Click the element you want to remove.

   A dialog box prompts you to confirm the removal.
3. Click **OK**.

Method 2:

1. Choose **Remove** from the pop-up menu for the element you want to remove.

   A dialog box prompts you to confirm the removal.
2. Click **OK**.

The element is removed from the workspace.

Specifying host names for business process nodes and servers

To specify the name of the host computer on which a business process node or server will run, perform the following steps:

1. Choose **Set Host** from the pop-up menu for a business process node or server.

   A dialog box prompts you for the name of the host computer. The default host name *localhost* is supplied in the dialog box.
2. Enter the host name or accept the default host name and press **Enter** or click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Showing server paths for business process nodes and servers

To show the server paths for a business process node or server, perform the following steps:

1. Choose **Show Server Paths** from the pop-up menu for a business process node or server.

   A dialog box displays the paths from the selected business process node or server to the server(s) that provide(s) the three types of services to the selected business process node or server. If you selected a server, and it supplies services to itself, the paths contain the selected server's name. If none of the three types of services is being provided to the selected business process node or server, the dialog box states that no path is specified.

2. Click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Creating backup nodes for servers

To create a backup node for a server, perform the following steps:

1. Choose **Edit Server Backup** from the pop-up menu for the server for which you want to designate a back up node.

   A dialog box displays default settings for a backup node. You can specify a host name, persistent storage name, communication protocol, and communication protocol values for the backup node. It is recommended that you specify a host name and a persistent storage name for the backup node that are different from those for the main server.

2. Enter the data for the backup node and click **OK**.

The dialog box closes, and a backup server symbol appears in the workspace. The symbol for a backup server is a shadow behind the server that is backed up.

Removing backup nodes for servers

To remove a backup node for a server, perform the following steps:

1. Choose **Remove Server Backup** from the pop-up menu for the server whose backup node you want to remove.

   A dialog box prompts you to confirm the removal.

2. Click **OK**.

The dialog box closes, and the symbol for the backup server is removed from the workspace.

Editing names of business process nodes and servers

When you create a business process node or server, the WFT generates a default name for it. You can edit the name of a business process node or server if you do not want to use the WFT default name.

To edit the name of a business process node or server, use either of the following methods:

Method 1:

1. Choose **Deployment Data>Node Name** from the pop-up menu for a business process node or server.

   A dialog box prompts you to edit the name of the selected business process node or server.

2. Enter the new name and click **OK**.

Method 2:

1. Choose either **Edit Text In Place** or **Edit Text** from the pop-up menu for the name of a business process node or server.

2. Enter the new name directly into the symbol for the business process node or server in the workspace, or enter it into the edit window that appears, depending on which command you chose.

The business process node or server name that is displayed in the workspace is changed to the new name.

Adding business process node users and passwords

Only business process nodes can have users; servers do not have users associated with them. To add user names and their associated passwords to a business process node, perform the following steps:

1. Choose **Deployment Data>Node User(s)** from the pop-up menu for a business process node.

   A dialog box displays the current list of user names for the selected business process node.

2. Enter the values for the *User Name* and *User Password* as appropriate and click **Add**.

   The name of the user you added is displayed in the *Node Users* list box in the dialog box.

3. Repeat step 2 for each user name you need to add and click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Note that to edit the information for a specific user in the *Node Users* list box in the dialog box, you must select a *User Name* from the list box, edit the values that appear in the *User Name* and *User Password* text entry lines in the dialog box, and then click **OK**.

Removing business process node users and passwords

Only business process nodes can have users; servers do not have users associated with them. To remove user names and their associated passwords from a business process node, perform the following steps:

1. Choose **Deployment Data>Node User(s)** from the pop-up menu for a business process node.

   A dialog box displays the current list of user names for the selected business process node.

2. Select from the *Node Users* list box in the dialog box the name of the user you want to remove.

   The values for *User Name* and *User Password* for the selected user name appear in the text entry lines in the dialog box.

3. Click **Remove**.

   The name you chose is removed from the *Node Users* list box.

4. Repeat steps 2 and 3 for each user name you need to remove and click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Specifying business process node or server arguments

To specify command-line arguments that are passed to a business process node or server at run time, perform the following steps:

1. Choose **Deployment Data>Node Arguments** from the pop-up menu for a business process node or server.

   A dialog box prompts you to enter the command-line arguments that you want to pass to the business process node or server at run time. If you have already entered arguments, they appear in the text entry dialog box.

2. Type the command-line arguments for the business process node or server and press **Enter** or click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Specifying persistent storage for business process nodes and servers

To specify persistent storage for a business process node or server, perform the following steps:

1. Choose **Deployment Data>Node Storage Name** from the pop-up menu for a business process node or server.

   A dialog box prompts you to enter the name of the persistent storage area.

2. Type the name of the persistent storage area and press **Enter** or click **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Showing business process node and server information

You can display certain information about business process nodes and servers within their symbols in the workspace. This information includes the name of the application that runs in the business process node or server (*Application Name*), the name of the host computer on which the business process node or server runs (*Host Name*), the name of the persistent storage location associated with the business process node or server (*Storage Name*), a list of user names specified for a business process node (*Users*), and a bitmap image (*Bitmap*). You can display any subset or all of this information at the same time.

To show information for business process nodes and servers, perform the following step:

➤ Choose **Application Name, Host Name, Storage Name, Users,** or **Bitmap** from the *Show* menu.

The *Application Name, Host Name, Storage Name, Users* (for business process nodes), and *Bitmap* (if specified) for each business process node and server are displayed in the symbols for the business process nodes and servers, depending on which commands you chose.

Building business process nodes and servers

You must build a business process node or server before you can run it.

To build a business process node or server, perform the following step:

➤ Choose **Deploy>Build** from the pop-up menu for a business process node or server.

The selected business process node or server is built.

Running a single business process node or server

Note that you must build a business process node or server before you can run it.

To run a single business process node or server from the Deployment Editor, perform the following step:

➤ Choose **Deploy>Run** from the pop-up menu for a business process node or server.

The WFT *wfenv messages* window displays the results of the request to run the selected business process node or server. The process of running the business process node or server first verifies the business process node or server, builds the application that runs in it, and generates its deployment information. The Deployment Editor then runs the business process node or server.

Running all business process nodes and servers

Note that you must build all the business process nodes or servers before you can run them.

To run all the business process nodes and servers from the Deployment Editor, perform the following step:

➤ Choose **Run All** from the *Actions* menu.

The WFT *wfenv messages* window displays the results of the request to run the business process nodes and servers. The process of running the business process nodes and servers first verifies the business process nodes and servers, builds the applications that run in them, and generates their deployment information. The Deployment Editor then runs all the servers and business process nodes.

Debugging a single business process node or server

Note that you must build a business process node or server before you can debug it.

To debug a single business process node or server from the Deployment Editor, perform the following step:

➤ Choose **Deploy>Debug** from the pop-up menu for a business process node or server.

The SNAP Debugger appears. You can debug your application as described in *Chapter 10, Using the Debugger,* in *Using the SNAP Development Environment*.

Cleaning persistent storage for a single business process node or server

Cleaning persistent storage for a business process node or server before running it prevents corruption of run-time data with any data lingering in persistent storage from a previous execution of the business process node or server.

To clean persistent storage for a single business process node or server, perform the following step:

➤ Choose **Deploy>Clean Storage** from the pop-up menu for a business process node or server.

Data are removed from the persistent storage area for the selected business process node or server.

Cleaning persistent storage for all business process nodes and servers

Cleaning persistent storage for business process nodes and servers before running them prevents corruption of run-time data with any data lingering in persistent storage from a previous execution of the business process nodes or servers.

To clean persistent storage for all business process nodes and servers, perform the following step:

➤ Choose **Clean Storage** from the *Actions* menu.

Data are removed from persistent storage for all your business process nodes and servers.

Showing communication link labels

You can display or hide the label associated with a communication link. The default label is the name of the communication protocol you specified for the communication link.

To show a communication link label, perform the following step:

➤ Choose **Show Label** from the pop-up menu for a communication link.

The communication link label appears on the communication link in the workspace. If you have not edited the label for the selected communication link, the default label appears. Otherwise, the label appears on the communication link as you have edited it.

Setting communication link protocol values

To set the communication protocol values for a communication link, perform the following steps:

1. Choose **Set Protocol Values** from the pop-up menu for a communication link.

   A dialog box prompts you to set the communication protocol values for the selected communication link.

2. Enter the data for the protocol values and press **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Setting communication link backup protocol values

To set the communication protocol values for a communication link to a backed-up server, perform the following steps:

1. Choose **Set Backup Protocol Values** from the pop-up menu for a communication link that connects a business process node or server to a server for which a backup server is specified.

   A dialog box prompts you to set the communication protocol values for the communication link to the backup server.

2. Enter the data for the protocol values and press **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Setting communication link protocols

To set the communication protocol values for a communication link, perform the following steps:

1. Choose **Set Protocol>(TCP/IP, UNIX Mail, or None)** from the pop-up menu for a communication link.

   If a business process node or server at either end of the communication link requires a new set of communication protocol values (as determined by the WFT), a dialog box prompts you to enter those protocol values for the selected communication link. You can specify whether the protocol values you enter apply to the server or to the business process node that is connected to the server. Whenever possible, default values are presented for the various data that you can enter.

2. Enter the data for the protocol values or accept the default values and press **OK**.

The dialog box closes, and you are returned to the Deployment Editor.

Moving workspace elements

You can move the following elements in the workspace: business process nodes, servers, communication links, and communication link labels. To move an element in the workspace, perform the following steps:

1. Select the **Select** tool.

2. Click and hold the left mouse button on the element you want to move.

3. Drag the element to a new location in the workspace, and release the mouse button.

   For communication links, dragging by the body of the symbol does nothing; dragging by the end-point selection handles moves the link, within the restriction noted below. For communication link labels, dragging the label moves it from one line segment in the selected communication link to another line segment in the same link.

The element is moved to the new location in the workspace. The other elements are adjusted, as necessary, to reflect the moved element's new location.

You can move the end points of communication links to new locations on their associated nodes, but you cannot move the links to different business process nodes or servers by dragging the link symbol from one business process node or server to another business process node or server.

You can move communication link labels from one line segment in a link to another line segment in the same link. However, you cannot move communication link labels from one communication link to another link.

CHAPTER

USING THE WORKFLOW SIMULATOR

About this chapter

This chapter describes the tools, menus, and other mechanisms provided by the Workflow Simulator for simulating and analyzing the operation of your workflow system.

Contents

Introduction

The **Workflow Simulator** enables you to create and run simulations of your workflow system design and deployment configurations. Using these simulations, you can evaluate alternative workflow system designs and deployment configurations, test "what if" scenarios to explore design contingencies and variations, and observe your design's performance characteristics.

See *Chapter 8, Simulating the Running of the WFT Workflow System*, in *Developing a WFT Workflow System* for more information about the concepts and process of building and running a workflow system simulation.

In this chapter, you will find how to:

- Build your workflow system for simulation by:
  - ▽ Defining simulation elements such as simulation nodes, tasks, and work item creators and processors.
  - ▽ Setting the timing of workflow system events.
  - ▽ Setting the probabilities of workflow system events.
  - ▽ Setting the number of instances for each simulation node.
  - ▽ Scheduling the activation and deactivation of simulation nodes over a given period of simulated time.
- Simulate the deployment configurations and the operation of your workflow system design.
- Perform "what if" analyses of workflow system design and deployment configuration variations.
- Observe the performance of the simulated operation of your workflow system design and deployment configuration.

Prerequisites

Before you use the Workflow Simulator, you must design your workflow system, using the Workflow Design Editor. See *Chapter 3, Using the Workflow Design Editor*, in this document and *Chapter 4, Designing a WFT Workflow System*, in *Developing a WFT Workflow System* for more information about designing your workflow system.

Important terms

The terms defined in Table 9-1 include some of the basic terms you need to understand to use the Workflow Simulator. More detailed definitions and definitions of additional terms are included in *Appendix A, The Workflow Template Glossary*, in *Developing a WFT Workflow System*.

Table 9-1. Important terms

Term	Definition
Animation	The graphic representation in the Workflow Simulator of the movement of work items through the workflow system during simulation.
Deployment	The process of defining the run-time configuration of a WFT workflow system by designating the relationships among the logical and physical elements of the system; the mapping of server and business process nodes to the physical architecture of the computer network.
Flow	A possible route between tasks through which a work item can travel. A flow is associated with a single type of work item or a work item set.
Junction	The point where one flow splits into multiple flows to form a copy flow or where multiple flows come together to form a compound flow. The junction for a compound flow can represent an *And* or an *Or* condition that is used to determine how work items are processed.
Simulation	The modeling of the operation of a workflow system design and its deployment configuration. Simulation uses mathematical modeling and other tools to simulate the movement and processing of work items among the tasks of the workflow system design. Simulation takes into account such variables as processing time delays, the probabilities of system events, and other factors. Simulation only models the flow of work items through tasks, rather than the task-specific processing that is applied to the work items in the tasks.
Simulation node	A partially configured node used by the Workflow Simulator to simulate deployed nodes of your workflow system. A simulation node is different from a node in the Deployment Editor; a simulation node is not configured or specified with deployment information as are nodes in the Deployment Editor. A simulation node is a collection of tasks, work item creators, and work item processors. Instances of simulation nodes are generated during a simulation.
Subworkflow	A graphic depiction of a subset of the overall business process described by a workflow design.
Task	The smallest significant unit of work activity within a business process; a point in the workflow where work items are created, processed, or destroyed.
Work item	The information processed by a task.
Work item creator	A Workflow Simulator element that creates simulated work items in a task within a simulation node according to a specified rate and probability.
Work item processor	A Workflow Simulator element that processes simulated work items in a task within a simulation node according to a specified speed. A work item processor can also create work items as a result of the processing.
Work item set	A collection of one or more work items delivered as input to a task.
Workflow system	A system that supports the automation of a subset or all of a business process. A workflow system automates the flow of work items throughout an enterprise, enabling the monitoring and controlling of the business process.

Accessing the Workflow Simulator

To access the Workflow Simulator, use either of the following methods:

Method 1:

➤ Click the Workflow Simulator button  in the WFT Development Environment main window.

Method 2:

➤ Choose **Workflow Simulator** from the *Editors* menu in the WFT Development Environment main window.

The Workflow Simulator appears, as shown in Figure 9-1.

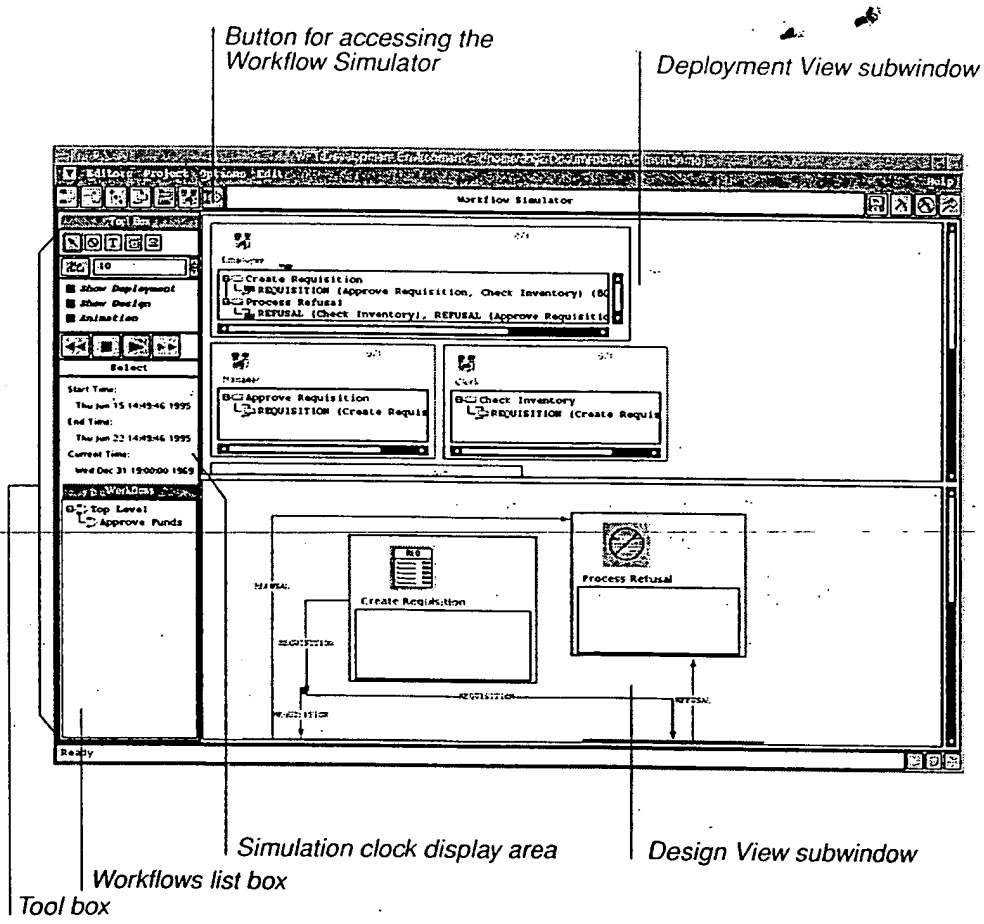


Figure 9-1. The Workflow Simulator

The Workflow Simulator provides subwindows, menus, and tools for simulating the deployment configuration and operation of your workflow system.

About the Workflow Simulator menus

The commands that appear on the *Edit* menu are listed in Table 9-2.

Table 9-2. Commands provided on the Workflow Simulator *Edit* menu

Command	Description
Set Workspace Size	Enables you to set the sizes of the workspaces in the *Deployment View* or the *Design View* subwindows. This command provides the following additional commands: • *Deployment View* – Enables you to manually change the size of the workspace in the *Deployment View* subwindow. • *Design View* – Enables you to manually change the size of the workspace in the *Design View* subwindow.
Auto Size Workspace	Automatically resizes the workspaces in the *Deployment View* and *Design View* subwindows to the smallest possible size that can contain all of the workspace display elements.

About the Workflow Simulator tool box

The Workflow Simulator tools, buttons, text entry line, *Workflows* list box, and subwindows are described in the following sections. Figure 9-2 shows an example of the Workflow Simulator tool box.

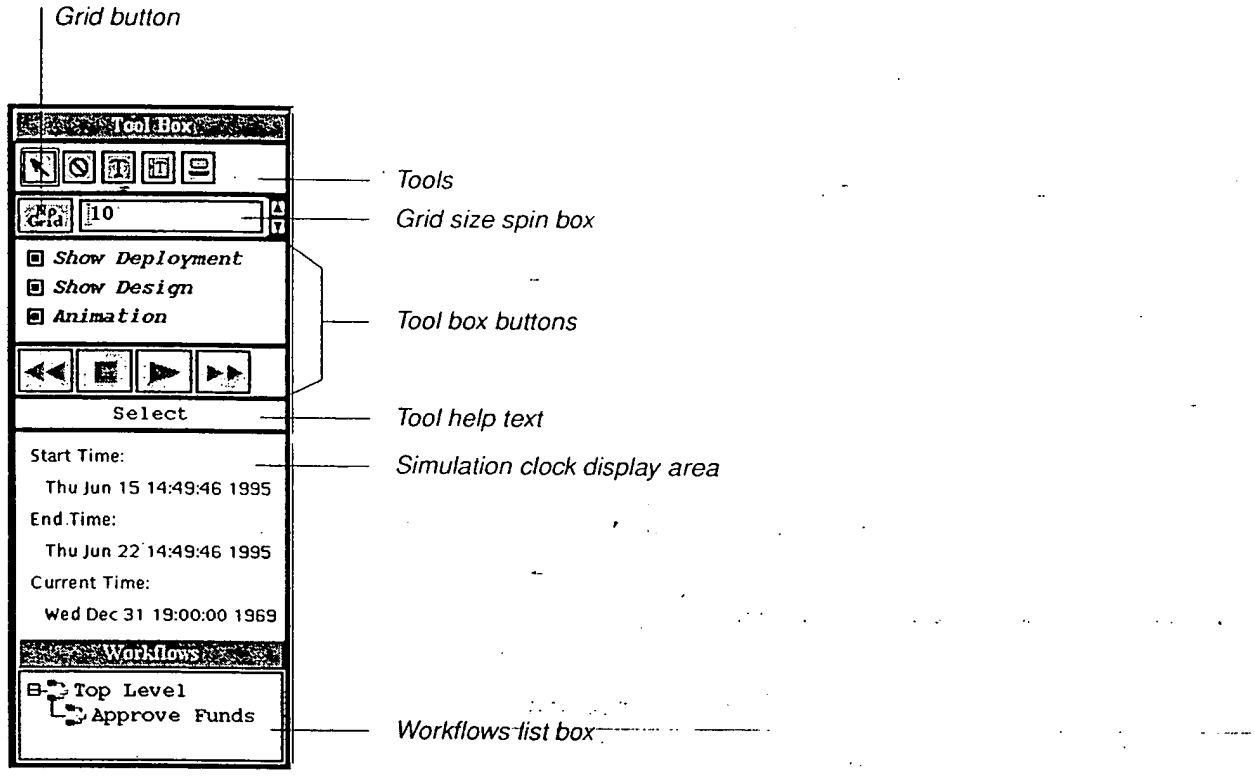


Figure 9-2. Example Workflow Simulator tool box

Workflow Simulator tools

The tools that appear in the tool box are listed in Table 9-3. In the tool box, the name of the currently selected tool is displayed in the tool help text area.

Table 9-3. Tools provided in the Workflow Simulator tool box

Tool	Description
 *Select*	Enables you to select items in the *Deployment View* and *Design View* subwindows.
 *Remove*	Removes simulation nodes from the *Deployment View* subwindow.
 *Edit Text In Place*	Enables you to edit simulation node names where they appear in the *Deployment View* subwindow.
 *Edit Text*	Enables you to edit simulation node names in the *Deployment View* subwindow in an edit window.
 *New Node*	Enables you to create simulation nodes in the *Deployment View* subwindow.

Workflow Simulator tool box buttons

The buttons that appear in the tool box are listed in Table 9-4.

Table 9-4. Buttons provided in the Workflow Simulator tool box

Button	Command
Grid options	Enables you to use the grid to help you place and align elements in the workspace. The grid is a ruled background on which you can snap elements to a particular location. The grid button provides the following options:
	*No Grid* – Turns the grid off, whether it is visible or not.
	*Grid On* – Turns the grid on and makes it visible. When the grid is on and visible, the grid button displays a graphic representation of a grid.
	*Grid Invisible* – Turns the grid on, but makes it invisible. When the grid is on, but invisible, the grid button displays a gray, shadow-like graphic representation of a grid.
Show Deployment	Displays the *Deployment View* subwindow.
Show Design	Displays the *Design View* subwindow.
Animation	Displays the animated work items in the *Design View* subwindow while the simulation is running.
Simulation control	Enable you to control the running of the simulation by starting, stopping, resetting, and stepping the simulation. The following buttons enable you to control the simulation:
	Resets the simulation to the beginning of the simulation period; clears all of the work item queues; and removes the status messages, queue workload bar charts, and animation, if any, displayed in the tasks in the *Design View* subwindow.
	Stops the simulation at the simulated current time; leaves all of the work item queues intact, and leaves all of the displayed information about the simulation intact. You are not required to stop the simulation before you edit simulation nodes, but doing so is recommended.
	Advances the simulation one step—that is, one simulation event; updates the queues and information displays; and updates the animation, if displayed.
	Runs the simulation from the current point to the end of the simulation period and continually updates the work item queues, information displays, and animation until the simulation ends.

Workflow Simulator tool box spin box

The spin box that appears in the tool box is listed in Table 9-5.

Table 9-5. Spin box provided in the Workflow Simulator tool box

Spin box	Description
Grid Size	Enables you to set the size of the grid squares, in pixels. Using the up and down arrows, you can increase and decrease the grid size within the range of values from 2 to 400, inclusive.

Workflow Simulator tool box simulation clock display area

The tool box simulation clock display area shows you the various times that are relevant to the simulation, including the *Start Time*, *End Time*, and the *Current Time*. The default values for these times are as follows:

- *Start Time* – The date and time when you first opened the Workflow Simulator in the current workflow system, which need not be the date and time of the current session.
- *End Time* – Exactly one week (seven days) after the *Start Time*.
- *Current Time* – *None*.

You can change the *Start Time* and the *End Time* by clicking their displays to invoke the Time Editor dialog box. You cannot change the *Current Time*; it changes from *None* to the simulated time when the simulation is running. When you stop a running simulation, the *Current Time* displays the simulated time as of the moment you stopped the simulation. When you reset a simulation, the *Current Time* displays *None*. See *Setting simulation start and end times* on page 9-27.

You set the various times in the Workflow Simulator according to your workflow system's view of real time. For instance, if you want to simulate the operation of your workflow system over the course of a real work week, the you specify the *Start Time* and *End Time* in terms of that real work week. You must enter the other various time values, including values for work item processing times, work item creation rates, polling intervals, elapsed times, and more, in terms of real times as well. For instance, if a task takes one minute of CPU time on your computer network to process a work item, then you enter one minute for that work item's real processing time. Similarly, if that same work item is delayed for nine minutes due to printing time, phone calls, in-basket waiting time, or some other reason, then you enter ten minutes (one minute of CPU time plus nine minutes of delay time) for that work item's elapsed time.

The Workflow Simulator uses its internal clock and timers to track all of the simulation's time-based events. As the simulation progresses, the Workflow Simulator processes one event after another, advancing the simulation clocks accordingly. Note that the simulation is not based on a scaled-down version of real time. Instead, the simulation is based upon event processing. For example, if your workflow system has a task that creates a work item once every five minutes, the simulation processes the creation of a work item, advances the clock to the next time-based event, which is the next work item creation in this example, and processes the next work item creation. The simulation does not sit idle for a scaled-down version of the five minutes between work item creations. This concept applies to all of the time values you specify for the simulation; each value you specify results in a time-based event that the Workflow Simulator processes.

Workflow Simulator tool box *Workflows* list box

The *Workflows* list box displays the names of all of the workflows currently defined for your WFT workflow system. You use this list box to display the different workflows in the *Design View* subwindow. Names in this list box that are displayed with a plus sign (+) beside them are workflows or subworkflows that contain other subworkflows. You double-click these names to expand or collapse the listing of the subworkflows within these workflows. You click the name of a workflow or subworkflow to display the workflow's contents in the *Design View* subwindow.

About the Workflow Simulator *Deployment View* subwindow

The *Deployment View* subwindow displays the configuration that is being simulated. Figure 9-3 shows an example of the *Deployment View* subwindow.

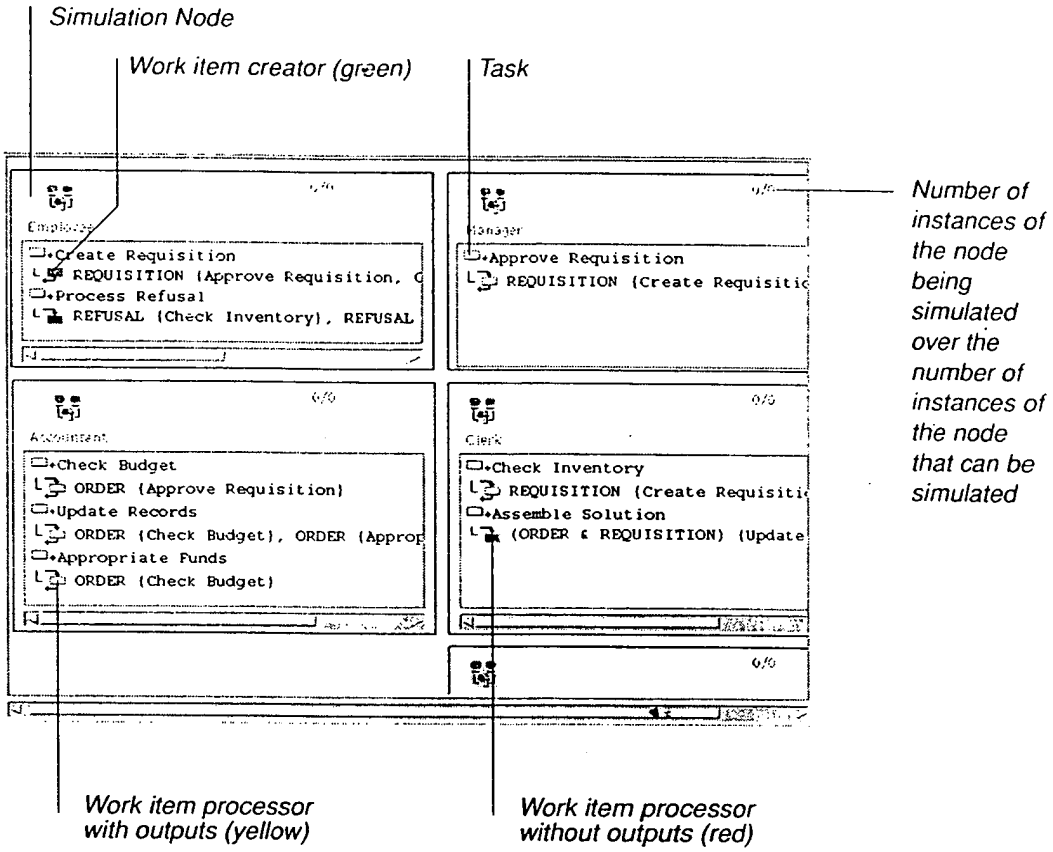


Figure 9-3. Example Workflow Simulator *Deployment View* subwindow

About the *Deployment View* subwindow pop-up menus

The *Deployment View* subwindow provides pop-up menus for simulation nodes, tasks in the nodes, and work item creators and processors associated with the tasks. These pop-up menus appear when you click a simulation node, task, or work item creator or processor using a non-left mouse button.

Pop-up menu for simulation nodes

The commands that appear on the pop-up menu for simulation nodes are listed in Table 9-6.

Table 9-6. Commands provided on the Workflow Simulator *Deployment View* subwindow pop-up menu for simulation nodes

Command	Description
Remove	Removes the selected simulation node from the workspace.
Edit Text In Place	Enables you to edit the name of the selected simulation node where it appears. This command is only available when you access the pop-up menu from a simulation node name.
Edit Text	Enables you to edit the name of the selected simulation node in an edit window. This command is only available when you access the pop-up menu from a simulation node name.
Set Tasks	Enables you to specify the tasks that are included in the simulation node.
Edit Polling Interval	Enables you to edit the polling interval for the selected simulation node. The default value is *5 Minutes*.
Set Bitmap File	Enables you to select a bitmap to be displayed within the symbol that represents the selected simulation node.
Do Not Use Bitmap File	Removes the bitmap from the selected simulation node and replaces it with the default bitmap.
Quantity	Enables you to specify the number of instances of the selected simulation node and the type of rate, flat or scheduled, at which the instances are to become active and inactive during the simulation time period. • *Flat* – Specifies that the simulation node instances are to become active at the beginning of the simulation period and remain active during the entire simulation period. A flat rate of 1 node instance is the default value for *Quantity*. • *Scheduled* – Specifies that the simulation node instances are to become active and inactive according to a schedule that you specify.

Pop-up menu for tasks within simulation nodes

The commands that appear on the pop-up menu for tasks within simulation nodes are listed in Table 9-7.

Table 9-7. Commands provided on the Workflow Simulator *Deployment View* subwindow pop-up menu for tasks within simulation nodes

Command	Description
New Creator	Enables you to create a work item creator in the selected task.
New Processor	Enables you to create a work item processor in the selected task.

Pop-up menu for work item creators and processors associated with tasks

The commands that appear on the pop-up menu for work item creators and processors that are associated with tasks are listed in Table 9-8.

Table 9-8. Commands provided on the Workflow Simulator *Deployment View* subwindow pop-up menu for work item creators and processors associated with tasks

Command	Description
Edit	Enables you to edit the selected work item creator or processor.
Remove	Removes the selected work item creator or processor from its associated task.

About the Workflow Simulator *Design View* subwindow

The *Design View* subwindow displays the design of the workflow system that is being simulated. This is the design you created using the Workflow Design Editor. This design represents the running simulation. During the simulation, the following information is displayed in the *Design View* subwindow:

- Bar charts depicting the work item queue load levels are displayed inside the task symbols.
- Symbols that represent the work items are shown moving along the flows, if animation is turned on.
- Messages that report the status of work item processing are displayed in the task symbols.

These bar charts, symbols, and messages are updated as the simulation progresses to show the loads in the queues, the movement of the work items, and the statuses of the tasks. Figure 9-4 shows an example of the *Design View* subwindow.

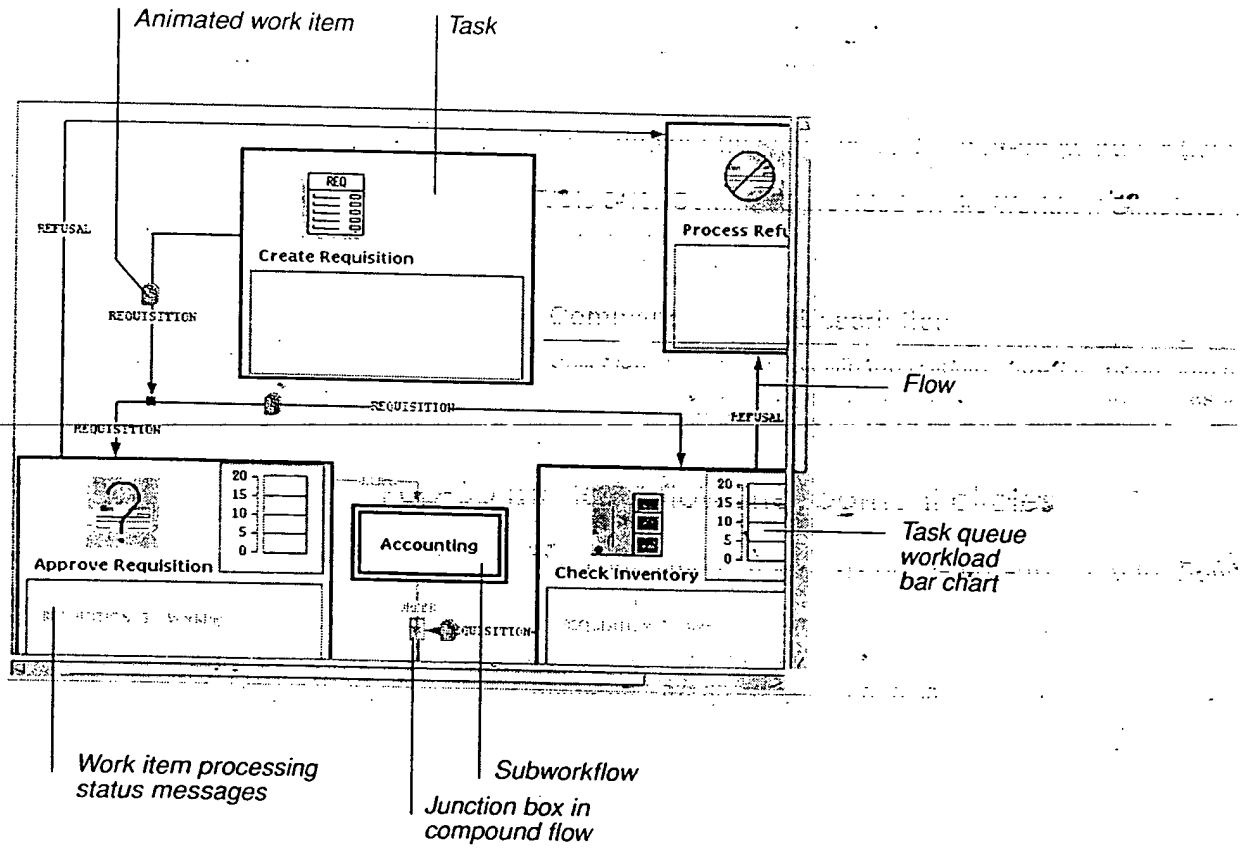


Figure 9-4. Example Workflow Simulator *Design View* subwindow

About the *Design View* subwindow pop-up menus

The *Design View* subwindow provides pop-up menus for tasks, flows, and flow line segment circles. These pop-up menus appear when you click a task, flow, or flow line segment circle using a non-left mouse button.

Pop-up menu for tasks

The command that appears on the pop-up menu for tasks are listed in Table 9-9.

Table 9-9. Command provided on the Workflow Simulator *Design View* subwindow pop-up menu for tasks

Command	Description
Create Node	Enables you to create a simulation node in the *Deployment View* subwindow in which you want the selected task to run, and enables you to specify the number of instances of the node you want to run during the simulation. This node and these instances are in addition to any other instances you specify in the *Deployment View* subwindow.

Pop-up menu for flows

The command that appears on the pop-up menu for flows is listed in Table 9-10.

Table 9-10. Command provided on the Workflow Simulator *Design View* subwindow pop-up menu for flows

Command	Description
Split Flow	Splits the selected flow into segments so that you can rearrange the graphic representation in the workspace.

Pop-up menu for flow line segment circles

The command that appears on the pop-up menu for flow line segment circles is listed in Table 9-11.

Table 9-11. Command provided on the Workflow Simulator *Design View* subwindow pop-up menu for flow line segment circles

Command	Description
Remove	Removes the selected flow line segment circle from the flow and joins the two adjacent flow line segments into a single flow line segment.

Creating, editing, and running simulations

This section provides step-by-step information about using the Workflow Simulator to construct and run a simulation of the operation and deployment configuration of your workflow system.

Creating simulation nodes

To create a simulation node in the *Deployment View* subwindow, use either of the following methods:

Method 1:

1. Select the **New Node** tool.
2. Click the point in the *Deployment View* subwindow where you want the simulation node to appear.

   A dialog box prompts you to select the tasks that you want to run in the simulation node you are creating. The list of tasks that is presented to you includes all of the tasks defined for your workflow system, regardless of the workflow or subworkflow in which they were defined.
3. Select the tasks you want to run in the simulation node you are creating and click **OK**.

   A new simulation node symbol that contains symbols for the tasks you selected appears in the *Deployment View* subwindow.

Method 2:

1. Choose **Create Node** from the pop-up menu for a task in the *Design View* subwindow.

   A dialog box prompts you to enter a value for the number of instances of a simulation node in which you want the selected task to run.
2. Type a value for the number of instances of a simulation node in which you want the selected task to run and press **Enter** or click **OK**.

   The dialog box disappears. A new simulation node symbol that contains the symbol for the selected task appears in the *Deployment View* subwindow. The node is defined to have a flat quantity of instances; that is, the node's *Quantity* is the number you specified and the rate is *Flat*. See *Specifying quantities of simulation node instances* on page 9-24 for more information.

Editing the assignment of tasks to simulation nodes

To edit the list of tasks assigned to a simulation node in the *Deployment View* subwindow, perform the following steps:

1. Choose **Set Tasks** from the pop-up menu for a simulation node.

   A dialog box prompts you to select the tasks that you want to run in the selected simulation node. The list of tasks presented to you includes all of the tasks defined for your workflow system, regardless of the workflow or subworkflow in which they were defined. The tasks that are currently selected to run in the node are highlighted.

2. Select the tasks you want to run in the selected simulation node and click **OK**.

The selected simulation node symbol containing symbols for the tasks you selected appears in the *Deployment View* subwindow.

Adding work item creators to tasks

To add a work item creator to a task in a simulation node in the *Deployment View* subwindow, perform the following steps:

1. Choose **New Creator** from the pop-up menu for a task in a simulation node.

   The Work Item Creation Editor appears. All of the work items that flow from the selected task, according to the design of your workflow system, are listed in the *Work Items* display area.

   This editor enables you to specify the following work item creation characteristics:

   - The simulated time interval between work item creations, including both the duration of the interval and the interval time units.
   - The probability of a work item being created during any given interval.
   - Any coincidental work item creations that occur in conjunction with the creation of the selected work item; this is necessary for a special case of attribute matching later in the workflow system. See *Specifying coincidental creations* on page 9-18 for further details.

2. Enter a number for *Interval*, which is the simulated interval of time between successive creations of the selected work item.

3. Choose a time unit from the pop-up menu for the *Interval* units display.

4. Enter a decimal number between zero and one, inclusive, for the *Probability*, click at the appropriate location on the *Probability* scale indicator, or drag the *Probability* scale indicator to the appropriate location.

5. Select the work items that you want to be created by the selected task.

6. Specify the coincidental creations, if any, for this work item creator, as described in the next section in this chapter.

7. Click **OK**.

The Work Item Creation Editor closes, and a work item creator symbol appears beneath the symbol of the selected task in the simulation node.

Specifying coincidental creations

You use coincidental creations to simulate special cases in workflow systems where domain data, which cannot be simulated, would be the basis for decisions concerning the combining of work items in compound flows. Coincidental creations are important to consider for workflow systems that contain compound flows whose input work items are not descendants of the same work item. The following is a brief discussion of how coincidental creations work.

The Workflow Simulator assigns an identifying number to every work item that is created during a simulation. If a task's processing of a work item results in the creation of another work item, the original work item's identifying number is attached to the newly created work item. In this sense, the newly created work item inherits the original work item's identifying number to show that it is a child of the original work item.

In the simulation of a compound flow, the work items that are combined into a work item set must have matching identifying numbers; that is, they must be offspring of the same original work item. Otherwise, the work items are not considered to match and cannot be combined. If two or more work items coming into a compound flow were created by unrelated ancestor work items, their identifying numbers do not match.

Coincidental creations provide a means for handling this situation. If you specify one work item to be a coincidental creation of another work item's creation, their identifying numbers will match, despite the fact that the work items were not created from the same ancestor work item.

When a work item creator generates a work item, that work item is assigned an identifying number. Also, if a coincidental creation is specified for that work item creator, the Workflow Simulator forces the tasks that contain the coincidental work items to create the work items and assign to them the same identifying number. Note that you can specify a time delay for the creation of coincidental work items to simulate the probable time delays of your real workflow system.

You can define a work item creator for a particular work item and specify coincidental creations for its coincidental work items. This models the case where one particular work item in the operating workflow system is always created before any of its coincidental work items. You can define work item creators that specify each other as coincidental creations; this models the case where either work item is created first in the operating workflow system. Note that work item creators that specify each other as coincidental creations do not result in an endless loop of one work item creating the other work item, and vice versa.

To specify coincidental creations for a work item creator in a task in a simulation node, perform the following steps:

1. Access the Work Item Creation Editor for a work item creator in which you want to specify coincidental creations, using either of the following methods:

   Method 1:

   ➤ Choose **New Creator** from the pop-up menu for a task in a simulation node.

   The Work Item Creation Editor appears for a new work item creator. In addition to those tasks you can perform with this editor, as described in *Adding work item creators to tasks* on page 9-17, you can specify coincidental creations in the new work item creator.

   Method 2:

   ➤ Choose **Edit** from the pop-up menu for the given work item creator in a task in a simulation node.

   The Work Item Creation Editor appears for the selected work item creator. In addition to those tasks you can perform with this editor, as described in *Adding work item creators to tasks* on page 9-17, you can specify coincidental creations in the existing work item creator.

2. Click **New** to the right of the *Coincidental Creations* display area in the Work Item Creation Editor.

   The Coincidental Creation Editor appears. All of the simulation nodes in the *Deployment View* subwindow are listed in the *Nodes* display area.

3. Enter a number for the *Delay Time*.

   The *Delay Time* is the period of time between the creation of the work item you are editing and the creation of the coincidental work item.

4. Choose a time unit from the pop-up menu for the *Delay Time* units display.

5. Select the simulation node that contains the task that contains the work item you want to be created coincidentally.

   The names of the tasks that run in the selected simulation node appear in the *Tasks* display area.

6. Select the task that contains the work item you want to be created coincidentally.

   The names of all of the work items that flow from the selected task, according to the design of your workflow system, are listed in the *Work Items* display area.

7. Select the work item that you want to be created coincidentally.

8. Click **OK**.

   The Coincidental Creation Editor disappears, and the work item appears in the *Coincidental Creations* display area in the Work Item Creation Editor.

9. Repeat steps 2 through 8 for each remaining coincidental work item creation for the work item creator that you are editing.

10. Click **OK**.

The Work Item Creation Editor disappears.

Adding work item processors to tasks

To add a work item processor to a task in a simulation node in the *Deployment View* subwindow, perform the following steps:

1. Choose **New Processor** from the pop-up menu for a task in a simulation node.

   The Work Item Processing Editor appears. All of the work items that flow into the selected task, according to your workflow design, are listed in the *Input Flows* display area.

   This editor also enables you to set a number of other work item processor characteristics. You can set the amount of time the task takes to process the work item, both in real time and in elapsed time. In the context of a simulation, real time is equivalent to the amount of dedicated CPU time that would be required to process the work item, while elapsed time is equivalent to real time plus any additional automation-related or end-user-related delaying time that is involved in the processing of the work item. You can specify any output work item creations that result from the processing of the selected work item, including the forwarding of the input flow work item to its next task. In specifying these output work item creations, you can set the work item to be created and its probability of creation.

   If you specify more than one output work item creation, you can specify that they are or are not exclusive. If you specify that output work item creations are exclusive, then at most one of the output work items is created when the selected work item is processed; the probabilities of all of the output work item creations are summed and are normalized to 100%. If you specify that output work item creations are not exclusive, each output work item creation is evaluated separately, according to its respective probability, to determine if the output work item is created when the selected work item's processing is complete.

2. Select the input flows that you want to the selected task to process.
3. Enter a number for *Real Time*.
4. Choose a time unit from the pop-up menu for the *Real Time* units display.
5. Enter a number for *Elapsed Time*.
6. Choose a time unit from the pop-up menu for the *Elapsed Time* units display.
7. Click the *Output Exclusive* check box to specify whether or not output work item creations are exclusive.
8. Specify the output work item creations for this work item processor, as described in *Specifying output work item creations* on page 9-21.
9. Click OK.

The Work Item Processing Editor closes, and a work item processor symbol appears beneath the symbol of the selected task in the simulation node. If you specified output work item creations, the work item processor symbol that appears is yellow and has an arrow entering the top of the box and another arrow exiting the bottom of the box. If you did not specify any output work item creations, the work item processor symbol that appears is red and has an arrow entering the top of the box, but no arrow exiting the bottom of the box.

Specifying output work item creations

In the Workflow Design Editor, you defined all of the flows of your workflow system. Therefore, you defined what work items flow from each task. In your workflow system design, you may have more than one type of work item flow from a particular task. The decision of which work item that task should create and send is made by the processing that occurs within the task. In an operating workflow system, at any given moment or in any given situation, a work item may or may not be created as a result of the processing that goes on in a task. During the simulation of the operation of your workflow system, output work items of a work item processor enable you to handle this situation by enabling you to specify how frequently a task creates a work item as a result of processing some input work item. This ability enables you to specify output work items in any way necessary to accurately reflect the nature of your workflow system.

To specify output work item creations for a work item processor in a task in a simulation node, perform the following steps:

1. Access the Work Item Processing Editor for a work item processor in which you want to specify output work items, using either of the following methods:

   Method 1:

   ➤ Choose **New Processor** from the pop-up menu for a task in a simulation node.

   The Work Item Processing Editor appears for a new work item processor. In addition to those tasks you can perform with this editor, as described in *Adding work item processors to tasks* on page 9-20, you can specify output work item creations in the new work item processor.

   Method 2:

   ➤ Choose **Edit** from the pop-up menu for the given work item processor in a task in a simulation node.

   The Work Item Processing Editor appears for the selected work item processor. In addition to those tasks you can perform with this editor, as described in *Adding work item processors to tasks* on page 9-20, you can specify output work item creations in the existing work item processor.

2. Click **New** at the right of the *Output* display area in the Work Item Processing Editor.

   The Work Item Processing Output Editor appears. The names of all of the work items that flow from the selected task, according to your workflow design, are listed in the *Work Items* display area.

3. Enter a decimal number between zero and one, inclusive, for the *Probability*, click at the appropriate location on the *Probability* scale indicator, or drag the *Probability* indicator to the appropriate location.

4. Select the work items you want to be created as output work items of the selected work item processor.

5. Click **OK**.

   The Work Item Processing Output Editor disappears. The output work item creation appears in the *Output* display area of the Work Item Processing Editor.

6. Repeat steps 2 through 5 for each remaining output work item for the work item processor that you are editing.

7. Click **OK**.

The Work Item Processing Editor disappears.

Editing work item creators and processors

To edit a work item creator or processor, use either of the following methods:

Method 1:

➤ Choose **Edit** from the pop-up menu for a work item creator or processor in a task in a simulation node.

Method 2:

➤ Double-click the work item creator or processor you want to edit.

Either the Work Item Creation Editor or the Work Item Processing Editor appears, depending on what you are editing. Edit the work item creator or processor, as necessary. See *Adding work item creators to tasks* on page 9-17 and *Adding work item processors to tasks* on page 9-20 for more information.

Editing simulation node polling intervals

When a task in a simulation node finishes processing a work item, it requests the next work item that is available to be processed. If such a work item is ready to be passed to the task, that work item is delivered to the task for processing. If such a work item is not ready, the task continues to make the request for the next available work item at regular intervals of time. This repetitive requesting is called **polling**. The **polling interval** is the time period between successive requests for the next work item.

To edit a simulation node's polling interval, perform the following steps:

1. Choose **Edit Polling Interval** from the pop-up menu for a simulation node.

   The Interval Editor appears, enabling you to set both the numeric value and the units for the polling interval.

2. Enter a number for the numeric value of the polling interval.

3. Choose a time unit from the pop-up menu for the polling interval units display in the Interval Editor.

4. Click **OK**.

The Interval Editor closes.

Specifying quantities of simulation node instances

The *Quantity* command on the pop-up menu that appears when you click a simulation node in the *Deployment View* subwindow enables you to specify the number of instances of the selected node that you want to be simulated and when those instances should become active and inactive during the simulation. The *Quantity* can be *Flat* or *Scheduled*.

The default for the *Quantity* command is *Flat*, with a default value of one node instance. This default means that only one instance of the selected simulation node will be simulated, and that the single instance will be active during the entire simulation period. You can specify that the *Quantity* be *Flat*, but that the simulation use more than one instance of the simulation node.

Alternatively, you can specify that the *Quantity* be *Scheduled*. That is, you can specify that the simulation use a specific number of instances (zero or more) of the simulation node, and that they become active and inactive according to a schedule that you specify. This ability enables the Workflow Simulator to model variable schedules that can more accurately reflect a real workflow system.

If you specify a schedule for a selected simulation node, the Workflow Simulator begins simulating at the beginning of the schedule. As node instances become active and inactive, according to the schedule, the Workflow Simulator starts running and stops running those instances of the simulation node. If the simulation period ends before the end of the schedule is reached, the Workflow Simulator does not process the end of the schedule. If the end of the schedule is reached before the end of the simulation period is reached, the Workflow Simulator cycles back to the beginning of the schedule and continues processing.

Note that the default length of your simulation node schedule is one week (seven days). You can increase or decrease the scheduled time period as you specify the schedule. You can set the length of the scheduled time period as described in this section and set the length of the simulation period as described in *Setting the simulation start and end times*. The scheduled time period of a simulation node and the overall simulation time period specified by the *Start Time* and *End Time* need not be the same length.

Specifying a flat number of node instances

To specify a simulation node's quantity as *Flat*, perform the following steps:

1. Choose **Quantity>Flat** from the pop-up menu for a simulation node.

   A dialog box prompts you to enter a value for the number of instances of the selected simulation node you want to run during the simulation.

2. Type or click the numeric buttons to enter a value for the number of instances of the selected simulation node you want to run during the simulation and press **Enter** or click either **OK** or **ENTER**.

The dialog box disappears.

Specifying a scheduled number of node instances

To specify a simulation node's quantity as *Scheduled*, perform the following steps:

1. Choose **Quantity>Scheduled** from the pop-up menu for a simulation node.

   A *Node Schedule* dialog box appears, enabling you to specify a schedule for the number of instances of the selected simulation node that you want to run during the simulation and the schedule for when you want them to become active and inactive.

   The dialog box enables you to enter several items of information in addition to the actual schedule. You can enter the *Maximum*, which is the maximum number of instances of the selected node that can be active at any given time during the schedule. The *Maximum* defines the vertical axis of the schedule graph.

   You can enter the *Variance*, which is the maximum number of instances of the node that the actual number of instances may vary from the number of instances specified in the schedule. The actual variance is calculated by the Workflow Simulator during the simulation according to a Gaussian curve.

   The actual number of instances of the node during the simulation will always be within the range of *zero* to *Maximum*. However, within that range, the actual number of node instances at any point in the schedule will be the specified number of instances according to the schedule, plus or minus the actual *Variance*. The *Variance* feature affords you more flexibility in modeling real workflow systems.

   You can enter both a numeric value and its time units for the *Duration* for the schedule. The *Duration* is the length of the scheduled period. You can enter both a numeric value and its time units for the *Interval* within the schedule. The *Interval* is the time increment between points on the schedule graph where you can specify a number of instances of the node to be active. Typically, the *Interval* time unit is smaller than the *Duration* time unit. The defaults are one week for the *Duration* and one hour for the *Interval*.

   The Workflow Simulator reevaluates the active number of instances of the node at every interval—that is, at every point in the schedule where you are allowed to specify a new value. A new actual variance and a new actual number of nodes are calculated at every interval.

   The schedule graph in the dialog box displays the following items:

   - The scheduled period (*Time*) on the horizontal axis.
   - The range of possible instances of the node (*Quantity*) on the vertical axis.
   - The specified number of instances of the node as a solid, dark (black) line across the graph.
   - The possible range of variance in the number of instances of the node as lighter (red) lines above and below the line for the specified number of instances of the node.

2. Enter a number for *Maximum*.

   The schedule graph changes, if necessary, to reflect the new *Maximum* value.

3. Enter a number for *Variance*.

   The schedule graph changes, if necessary, to reflect the new *Variance* value.

4. Enter a number for *Duration*.

   The schedule graph changes, if necessary, to reflect the new *Duration* value.

5. Choose a time unit from the pop-up menu for the *Duration* units display.

   The schedule graph changes, if necessary, to reflect the new *Duration* units.

6. Enter a number for *Interval*.

   The schedule graph changes, if necessary, to reflect the new *Interval* value.

7. Choose a time unit from the pop-up menu for the *Interval* units display.

   The schedule graph changes, if necessary, to reflect the new *Interval* units.

8. Click and hold the left mouse button on a point along the dark line on the graph that represents the specified number of instances of the selected node.

9. Drag the cursor up or down to the point on the graph that corresponds to the number of node instances that you want to run during that particular interval of the schedule, and release the mouse button.

   The point on the line representing the number of instances of the node during the selected interval is moved to the new value.

10. Repeat step 10, as necessary, for each interval along the line in the schedule graph.

11. Click **OK**.

The *Node Schedule* dialog box disappears.

Setting simulation start and end times

The default settings for *Start Time* and *End Time* result in a simulation period that covers seven days. You can set the *Start Time* and *End Time* to be whatever you require for your simulation. The Workflow Simulator begins simulating at the beginning of your node schedule and continues for as much of your schedule as can be processed between the *Start Time* and *End Time*. If your node schedule is longer than the simulation time period, the simulation does not process all of your node schedule. If your node schedule is shorter than the simulation time period, the simulation cycles through your node schedule until the simulation time period ends.

To set the simulation *Start Time* and *End Time*, perform the following steps:

1. Click the display of the *Start Time* in the simulation clock display area.

   The Time Editor appears, enabling you to set the month, date, hours, minutes, seconds, and year for the *Start Time*.

   Arrows are displayed both above and below the various elements of the *Start Time* display in the Time Editor. Clicking the arrows above the elements increases the values of the elements. Similarly, clicking the arrows below the elements decreases the values of the elements. Double-clicking the arrows sets the value to the nearest logical beginning or ending point for the specific unit; for example, double-clicking the arrow below the hours sets the hours to *00*, while double-clicking the arrow above the hours sets the hours to *23*.

2. Click the arrows above and below the various elements of the *Start Time*, as necessary, to set the appropriate date and time.

3. Click **OK**.

   The Time Editor closes, and the *Start Time* is displayed with the new values.

4. Repeat steps 1 through 3 for the *End Time* in the simulation clock display area.

5. Click **OK**.

The Time Editor closes, and the *End Time* is displayed with the new values.

Note that you cannot set the *Current Time* for the simulation.

Stepping simulations

Stepping a simulation means to advance the simulation one step at a time. A simulation step is considered to be one simulation event. For example, a simulation event might be a creation or processing event for a work item or task.

To step a simulation, perform the following steps:

1. Click the  button in the tool box.

2. Repeat step 1, as necessary.

The simulation advances one step, or one event, at a time. The *Current Time* value in the simulation clock display area displays the current simulation time at the end of the simulation step. The numbers displayed in the upper right corners of the simulation node symbols in the *Deployment View* subwindow display the number of instances of that node that are active at the end of the step. Work item queue bar charts in the tasks displayed in the *Design View* subwindow display the work item workload level at the end of the step. Symbols representing work items appear on the flows between tasks in the *Design View* subwindow, if the animation is turned on.

If you are using the Motif tool set, you can step a simulation by positioning the mouse cursor over the  button and pressing the space bar. The result is exactly the same as for the above procedure. Holding the space bar down is the same as repeatedly clicking the button to step the simulation. If you are using MS Windows or OS/2, you can click the button, which places the focus of the windowing system on the button, and then press the space bar to achieve the same result.

Running simulations

Running a simulation means to advance the simulation continuously until either it ends or you stop it manually.

To run a simulation, perform the following step:

➤ Click the  button in the tool box.

The simulation advances continuously until either you stop the simulation or the simulation time period expires. The *Current Time* value in the simulation clock display area displays the current simulation time as the simulation runs. The numbers displayed in the upper right corners of the simulation node symbols in the *Deployment View* subwindow display the number of instances of that node that are active as the simulation runs. Work item queue bar charts in the tasks displayed in the *Design View* subwindow display the work item workload level as the simulation runs. Symbols representing work items appear on the flows between tasks in the *Design View* subwindow, if the animation is turned on.

Stopping simulations

To stop a simulation while it is running, perform the following step:

➤ Click the [■] button in the tool box.

The simulation stops running. The *Current Time* value in the simulation clock display area displays the simulation time as of when you stopped the simulation. The numbers displayed in the upper right corners of the simulation node symbols in the *Deployment View* subwindow display the number of instances of that node that are active as of when you stopped the simulation. Work item queue bar charts in the tasks displayed in the *Design View* subwindow display the work item workload level as of when you stopped the simulation. Symbols representing work items appear on the flows between tasks in the *Design View* subwindow, if the animation is turned on.

Resetting simulations

To reset a simulation to the beginning of its simulation time period, perform the following step:

➤ Click the [◄◄] button in the tool box.

The simulation is reset. The *Current Time* value in the simulation clock display area displays the default value *None*. The numbers displayed in the upper right corners of the simulation node symbols in the *Deployment View* subwindow are reset. Work item queue bar charts in the tasks displayed in the *Design View* subwindow disappear. Symbols representing work items disappear from the flows between tasks in the *Design View* subwindow, if the animation is turned on.

Moving workspace elements

To move an element in either of the Workflow Simulator subwindows, perform the following steps:

1. Select the **Select** tool.

2. Click the workspace element you want to move.

   Handles appear on the workspace element to indicate that it is selected.

3. Drag the element to the new location in the workspace and release the mouse button.

   For tasks and subworkflows in the *Design View* subwindow and for simulation nodes in the *Deployment View* subwindow, dragging by the body of the symbol moves the element; dragging by the selection handles resizes the element. For flows in the *Design View* subwindow, dragging by the body of the symbol does nothing; dragging by the end-point selection handles moves the element, within the restriction noted below.

The element is moved to the new location in the workspace. The other elements are adjusted, as necessary, to reflect the moved element's new location.

You can move the end points of flows to new locations on their associated tasks, but you cannot move the flows to new tasks by dragging the flow symbols from one task to another task.

Removing workspace elements from the *Deployment View* subwindow

Because the *Design View* subwindow reflects the design of your workflow system as specified in the Workflow Design Editor, you cannot remove elements from or add elements to the *Design View* subwindow, with the single exception of the flow line segment circles that are only used for layout purposes.

You can, however, add and remove elements in the *Deployment View* subwindow. Previous sections in this chapter describe how to add elements. You can remove simulation nodes, tasks in simulation nodes, and work item creators and processors associated with the tasks.

Removing simulation nodes

To remove a simulation node from the *Deployment View* subwindow, use either of the following methods:

Method 1:

1. Select the **Remove** tool.
2. Click the simulation node you want to remove.

   A dialog box prompts you to confirm the removal.
3. Click **Yes**.

Method 2:

1. Choose **Remove** from the pop-up menu for the simulation node you want to remove.

   A dialog box prompts you to confirm the removal.
2. Click **Yes**.

The simulation node is removed from the *Deployment View* subwindow.

Removing tasks from simulation nodes

To remove a task from a simulation node in the *Deployment View* subwindow, perform the following steps:

1. Choose **Set Tasks** from the pop-up menu for the simulation node that contains the task you want to remove.

   A dialog box prompts you to select the tasks that you want to run in the selected simulation node. The tasks that are already selected are highlighted in the dialog box.
2. Select the task you want to remove from the simulation node.

   The selected task is deselected, and its highlighting is removed.
3. Click **OK**.

The task is removed from the simulation node in the *Deployment View* subwindow.

Removing work item creators and processors from tasks

To remove a work item creator or processor from a task in a simulation node in the *Deployment View* subwindow, perform the following step:

➤ Choose **Remove** from the pop-up menu for the work item creator or processor you want to remove.

The work item creator or processor is removed from the task in the simulation node in the *Deployment View* subwindow.

Resizing task, subworkflow, and simulation node symbols

To resize a task or subworkflow symbol in the *Design View* subwindow or a simulation node symbol in the *Deployment View* subwindow, perform the following steps:

1. Select the **Select** tool.
2. Click the task, subworkflow, or simulation node symbol you want to resize.

   Handles appear on the task, subworkflow, or simulation node symbol to indicate that it is selected.

3. Drag any of the selection handles to resize the task, subworkflow, or simulation node symbol to the desired dimensions, and release the mouse button.

The resized task, subworkflow, or simulation node symbol appears in the workspace.

APPENDIX

THE WFT PROJECT STRUCTURE

About this appendix

This appendix describes the WFT project file structure for a workflow system.

Contents

Introduction

A number of files support the WFT Development Environment and your run-time workflow system. At run time, the WFT creates additional files that hold various types of run-time data. These files are organized into hierarchical directories that correspond to the types of tasks and applications in your workflow system and to the nodes you create from those applications. A one-to-one correspondence exists between the directories and each type of task, application, and node in your workflow system. Figure A-1 shows the directory structure of a WFT workflow system.

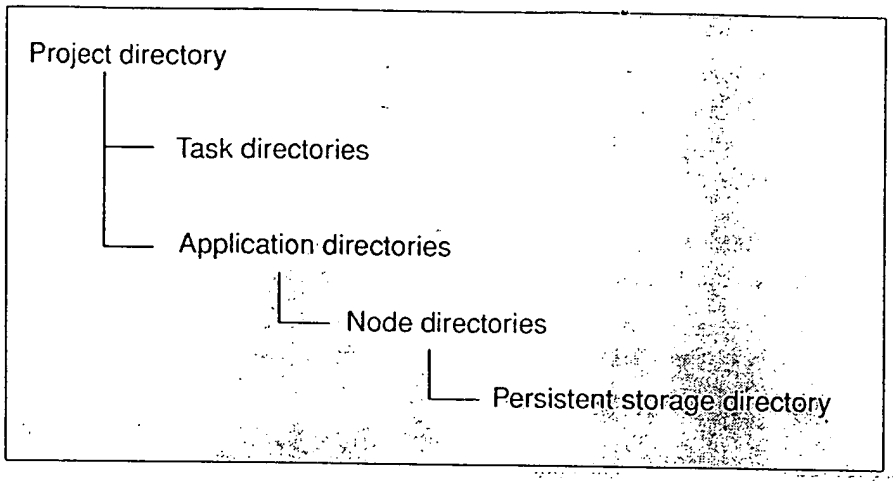


Figure A-1. Project directory structures

The main directory for a WFT workflow system is called the **project** directory. All files and subdirectories created for your workflow system reside in the project directory or in one of its subdirectories. Figure A-1 illustrates the basic directory structure of the project directory.

This appendix describes the structure and contents of the project directory and its subdirectories.

About WFT Development Environment files

You should not modify most of the files created by the WFT Development Environment. The exceptions are the class definition (*cd*) files such as *wfsys.cd* and *route.cd*. These files contain your domain code. If you modify these files outside of the WFT Development Environment, the WFT will incorporate the changes you make when you invoke the WFT Development Environment. If you modify these files outside of the WFT Development Environment while you have the WFT running, the WFT cannot know that you are making changes. In this case, you must rebuild your base parsed class definition (*pcd*) file in the WFT Development Environment after you are finished editing the *cd* files.

About the project directory

The project directory is the main directory for your workflow system. It contains most of the WFT Development Environment files and your workflow system's main cd file, *wfsys.cd*. Table A-1 describes the files that the WFT Development Environment creates in the project directory.

Table A-1. Files created by the WFT Development Environment in the project directory

File name	Description
aped.dat	Application Editor information
classes.er	Layout information for the Class Definition Editor
deped.dat	Deployment Editor information
design.dat	Workflow Design Editor run-time information
formdef.dat	Form Editor information
route.cd	Workflow Design Editor routing rules
scod.dat	Schema information for non-generated schemas
schmdef.dat	Schema Editor information
schmrel.dat	Work item and schema mapping information
server.cd	Abbreviated class definitions for inclusion in servers
simulatn.log	Workflow Simulator session information
tsked.dat	Task Editor information
wded.dat	Workflow Design Editor layout information
wfenv.dat	Workflow project information
wfproj.loc	Local project information, including paths
wfsys.cd	Main class definition file
wfsys.pcd	Parsed main class definition file
wfsys.sim	Workflow Simulator information

About the task directories

A task directory exists for each type of task you create using the WFT Development Environment. When you create a task, its task directory contains the files created by the WFT Development Environment. Table A-2 describes the files that the WFT Development Environment creates in each task directory.

Table A-2. Files created by the WFT Development Environment in the task directories

File name	Description
tdsk.dat	Default task desk information
formrel.dat	Work item and form mapping information
wftaed.dat	List of files needed to edit the task

Once you edit a task using the Task Editor, the standard data files that make up a SNAP application, such as *macro.dat* and *target.dat*, are added to the directory.

About the application directories

An application directory exists for each application you create using the WFT Development Environment. When you create an application, its directory contains four files created by the WFT Development Environment. Table A-3 describes the files that the WFT Development Environment creates in each application directory.

Table A-3. Files created by the WFT Development Environment in the application directories

File name	Description
formrel.dat	Work item and form mapping information
task.dat	Task-related data file. This file exists when a task included with an application has been edited.
tdsk.dat	Default task desk information
wftaed.dat	List of files needed to edit the application

Once you edit an application using the Application Editor, or edit any task grouped within the application, the standard files that make up a SNAP application are added to the directory.

About the business process node and server directories

A business process node or server directory exists for each business process node or server you create and build using the WFT Development Environment. When you create a business process node or server, its directory contains three files created by the WFT Development Environment. Table A-4 describes the files that the WFT Development Environment creates.

Table A-4. Files created by the WFT Development Environment in the business process node or server directories

File name	Description
inst.dat	Workflow deployment information
user.dat	The business process node or server's user definitions
workflow.dat	Business process node or server data file information

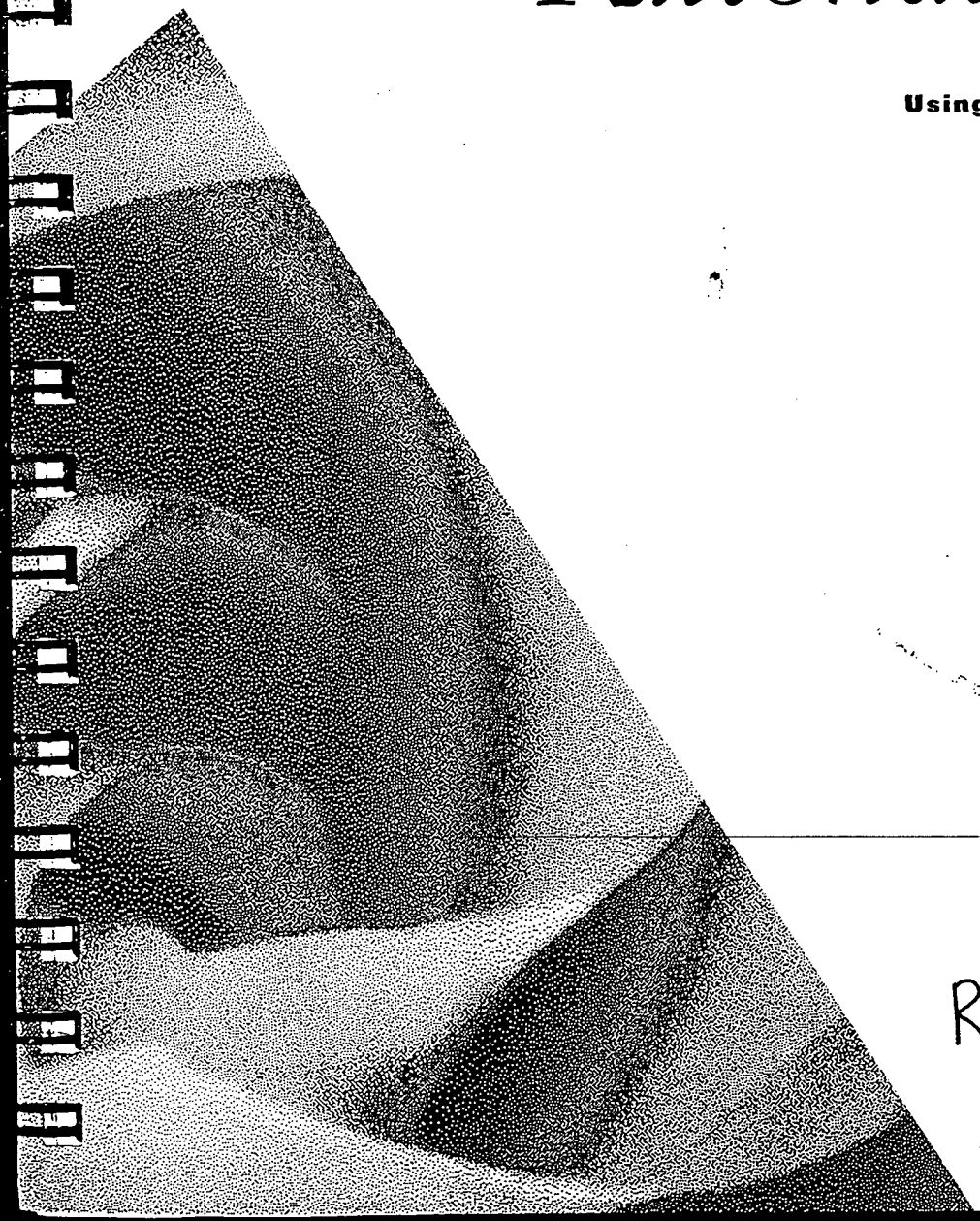
About the persistent storage locations

A persistent storage location exists for each business process node or server you run. This location is removed when you choose the *Clean Storage* command in the Deployment Editor. You should not edit or manipulate in any way the contents of the persistent storage locations.

INDEX

Rational Rose

Using Rational Rose 4.0



RAT-UR

RATIONAL
SOFTWARE CORPORATION

Using Rational Rose 4.0

Part Number 505-006786-000

Revision 4.0 November 1996 (Software Release 4.0)

This document is subject to change without notice.

Note the Reader's Comments form at the end of this book, which requests your evaluation to assist Rational in preparing future documentation.

Rational Software Corporation
2800 San Tomas Expressway, Santa Clara, CA 95051-0951

Chapter 10

Team Development

Overview

To support teams of analysts, architects, and engineers practicing controlled iterative development, the software:

- Enables each developer to operate in a private workspace containing the full model, with exclusive control over the propagation of changes into that workspace.
- Supports decomposition of the model into controlled units and integrates with any project-wide Configuration Management (CM) system to maintain the integrity of those controlled units.
- Uses platform-independent model files for persistent storage of controlled units, and provides the path map mechanism to allow model files to be moved or copied among workspaces and archives.

Controlled Iterative Development

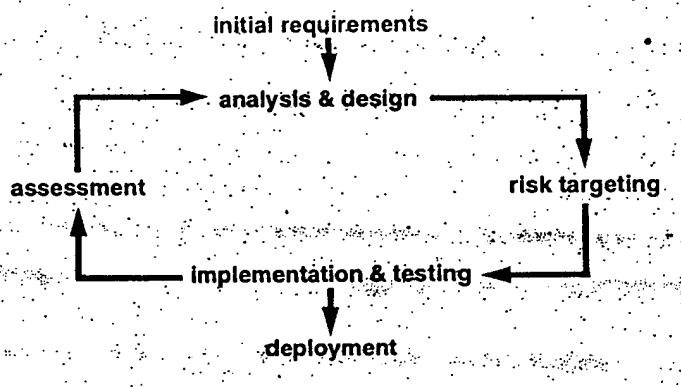
In the controlled iterative process, development is carried out through a sequence of iterations. Each iteration begins with an assessment of the current analysis, design, and implementation to identify critical, unresolved risks. The most critical unresolved risks are used to drive the iteration. Scenarios illustrating the risks are identified, and the current analysis and design are extended to address the risks.

To provide objective proof that the targeted risks have been retired, the current implementation is extended—as minimally as possible—and tests organized around the risk scenarios are constructed. It is common for the design to

evolve during extension of the implementation—for example, new operations may be added, or several classes may be relocated from one logical package to another to strengthen an abstraction.

When the extended implementation successfully executes all tests, the implementation is reviewed and, if acceptable, the analysis and design model is updated to reflect the design changes instituted in the implementation. When this update is complete, the next iteration proceeds.

The controlled iterative development process is shown graphically below:



Setting Up Multiuser Iterative Development

This section provides step-by-step instructions for setting up a multiuser development, and executing an iteration. The process implied by these steps is representative, with many variations possible to support your specific process. In particular, this process assumes that each controlled unit is owned by one developer—a developer can own several controlled units, but no controlled unit is owned by more than one developer. The process further assumes it is financially worthwhile to allocate to each developer sufficient filesystem storage to maintain in their workspace a full copy of all controlled units. Both of these assumptions can be relaxed if appropriate.

Subsequent sections of this chapter provide additional information about controlled units, the path map mechanism, and integration with a configuration management system.

To set up multiuser development with Rose:

1. Select a filesystem directory to represent each developer's workspace, and choose a virtual path symbol—such as **$mywork**—to represent the root of the workspace directory hierarchy. Create a path map entry in each developer's .ini file; this entry should map the virtual path symbol **$mywork** to the actual pathname of the user's workspace directory:

   ```
   [Virtual Path Map]
   mywork=c:\project\username\workspce
   ```

   With this setup, Rose will construct $mywork-relative filenames in all model files, allowing these model files to be moved or copied among workspaces and archives.

   Rose updates its path map with changes made to the .ini file whenever **Open** from the **File** command is executed.

2. Select a filesystem directory to represent the integration workspace, and setup a .ini file with the path map entry:

   ```
   [Virtual Math Map]
   mywork=c:\project\intgrtn\workspce
   ```

   Use this .ini file whenever you are operating in the integration workspace.

3. Establish a script and a Rose CM menu entry for each configuration management operation, such as **Control**, **CheckOut**, **CheckIn**, and **AcceptChanges**. The software includes basic scripts for use with a configuration management system popular on your platform; you can use these scripts "as is," extend them, or use them as a guide when creating scripts for use with your configuration management system. Additional information is provided in the section of

this chapter entitled *Integrating With a Configuration Management System.*

4. Create an initial model in the integration workspace, and decide which logical and component packages in this model represent appropriate elements for parallel development. Use Rose's **File:Units:Control** command to designate each such element as a controlled unit, and create its associated model file (.cat file) in your integration workspace directory. The model itself is also an element to be controlled. Use Rose's **File:Units:CM** command to direct your configuration management system to control each controlled unit.

   Set the integration workspace model's Design-level Directory code generation property to the source code directory within the integration workspace. The name of this source code directory should be specified using the virtual path symbol **$mywork**. Generate code from the initial model to establish an implementation directory hierarchy within the integration workspace. Rose will generate a physical model from the logical package hierarchy if the initial model does not contain component packages. You may also wish to establish integration workspace directories for requirements, specifications, plans, documentation, and tests.

   Direct your configuration management system to control each source code unit, as well as specifications, plans, documents and tests you deem appropriate; these latter artifacts are referred to as auxiliary units.

   In the integration workspace, create a Project for the Rational/C++ Analyzer, specifying the Base Projects, Directories, Extensions, Files, and Export Options required to Reverse Engineer your source code during each iteration. Additional information is provided in the *Round Trip Engineering with Rational Rose 4.0* manual.

5. Capture this initial state by directing your configuration management system to construct a release of all units it controls. Initialize the developer

workspaces by copying the integration workspace directory hierarchy and files to each developer working directory.

Practicing Iterative Development

After completing the five steps described in the previous section, your team can begin an iteration. The typical iteration will proceed through six phases:

Planning In the integration workspace:

- Identify risks.
- Establish iteration objectives and schedules.
- Capture new scenarios and plan their tests.

Propagation In the integration workspace:

- Propagate controlled units from the integration workspace into each developer workspace.
- Propagate implementation source code and auxiliary units from the integration workspace into each developer workspace.

Extension In the developer workspaces:

- Extend the current analysis and design.
- Extend the current implementation source code.
- Extend the current scenario tests, and verify their successful execution.

Integration In the integration workspace:

- Accept all modified controlled units into the integration workspace.
- Accept all modified implementation source code units and auxiliary units into the integration workspace.
- Identify and resolve inconsistencies.

- Verify successful execution of all scenario tests and regression tests.

Assessment In the integration workspace:

- Identify and validate design changes introduced during implementation.
- Assess iteration results vs. established objectives.
- Update analysis and design model to reflect design changes.

Release In the integration workspace:

- Create a comprehensive iteration release.

Planning Phase

In the Planning phase, use interaction diagrams to capture the scenarios that drive the iteration.

Propagation Phase

In the Propagation phase, use the **File:Units:CM** command to direct your configuration management system to AcceptChanges from each controlled unit, source code unit, and auxiliary unit in the integration workspace to each developer workspace. You can create a PropagateAll script that automates this step if desired. If your configuration management system does not automatically construct directories when it accepts changes, create these directories manually.

Extension Phase

During the Extension phase, each developer operates in his/her own workspace, loading those controlled units they need to reference or modify. Before modifying a controlled unit, a developer must use the **File:Units:CM** command to CheckOut that unit. Rose's write-protection mechanism prevents a developer from attempting to modify a controlled unit that has not been successfully checked out.

Developers can use Rose Code Generation to support source code extension for model components they create in his/her workspaces; after generating code for new components, direct the configuration management system to control the new source code units. If an existing class is re-assigned from one module to another, relocate any existing source—using the configuration management system as appropriate—before generating code for the class.

If one developer encounters the need to modify a model component contained in a controlled unit assigned to another developer, there are two basic procedures that can be used, depending on your process:

- The developer needing the change makes the change: check-out the controlled unit, make the changes in your workspace, and check the unit back in; the unit's owner will have to AcceptChanges from your workspace before checking the unit out.
- The developer owning the controlled unit makes the change: ask the owner to make the changes in their workspace and check-in the controlled unit; then AcceptChanges into your workspace.

Developers can reverse engineer the source code in their workspace using the Rose Analyzer's **FirstLook** or **DetailedAnalysis** Export Option Set, and examine the resulting model with Rose. When satisfied, you can reverse engineer the source code with the Analyzer's **RoundTrip** Export Options Set, and use Rose's **File:Merge** command to update your workspace model for consistency with its source code. Be sure to inspect the log entries produced by the merge.

Each developer should use the **Tools:CheckModel** command to identify inconsistencies within their workspace; all such inconsistencies must be eliminated before proceeding to Integration. If necessary, a developer can update his/her workspace with the most-recently checked-in version of a controlled unit by using the **File:Units:CM** command to execute the AcceptChanges command.

Integration Phase

In the Integration phase, load all controlled units in the integration workspace, and AcceptChanges for each unit modified in the iteration; you can create an AcceptAll script that automates this step if desired. Use Rose's **Tools:CheckModel** command to identify inconsistencies introduced by the integration of units extended in parallel. Accept all modified source code units and auxiliary units from developer workspaces. If your configuration management system does not automatically construct directories when it accepts changes, create these directories manually. Verify successful execution of all scenario and regression tests before proceeding to Assessment.

Assessment Phase

In the Assessment phase, reverse engineer the source code in the integration workspace using the Rose Analyzer's **FirstLook** or **DetailedAnalysis** Export Option Set, and examine the resulting model with Rose. Use **Tools:ShowDifferences** to reveal design changes by comparing the reverse engineered model with the integration workspace model. Additional information about model differencing is provided in the *Model Differencing* chapter of this book.

When you are satisfied with the iteration, reverse engineer the source code with the Analyzer's **RoundTrip** Export Options Set, and use Rose's **File:Merge** command to update the integration workspace model for consistency with its source code. Inspect the log entries produced by the merge, and use the **Tools:CheckModel** command during your final review of the model.

Release Phase

In the Release phase, direct your configuration management system to construct a release of all units it controls—requirements, specifications, plans, controlled units in your model, source code, tests, and documentation.

Controlled Units

Using the **File:Units:Control** command in the **File** menu's units shortcut menu, or the **Control** button in the Units dialog box displayed by the **Browse:Units** command, the following model components can be controlled:

- The model kernel—the connective framework for all other model subunits. (The model kernel is always a unit.)
- Any logical package in the model.
- Any component package in the model.
- The model's deployment diagram.

If the **Options:Display:UnitAdornment** option is enabled, icons representing controlled units will be adorned with an octagon containing the letter **U**.

Storing Controlled Units

Controlled units are stored in model files, using specific filename extensions to designate their contents:

- The file containing the model kernel utilizes the filename extension **.mdl.**
- Files containing controlled units which are the model's logical packages utilize the filename extension **.cat.**
- Files containing controlled units which are the model's component packages utilize the filename extension **.sub.**
- The file containing the controlled unit which is the model's deployment diagram utilizes the filename extension **.prc.**
- The file containing controlled units which are the property sets for the model utilizes the filename exension **.prp**. This file is indirectly modified throught the Rose property editors.

Loading and Unloading Controlled Units

When a model containing controlled units is first opened, Rose presents a dialog box asking if all units should be immediately loaded. If the model is large, or you are planning to work on a few specific units, you can reduce resource consumption by clicking **No** and then loading units selectively.

Rose maintains references between components in unloaded units and components remaining in the model. These references are automatically resolved when the unloaded units are subsequently loaded.

If the **Options:Display:UnresolvedReferenceAdornments** option is selected, icons representing components in unloaded controlled units, and icons representing relationships involving an unloaded controlled unit are adorned with a small octagon containing the letter **M** overstruck with a slash.

Units can be loaded and unloaded using the **File:Units:Load** and **File:Units:Unload** commands respectively, or using the **Load** and **Unload** buttons in the Units dialog box displayed by the **Browse:Units** command. An unloaded controlled unit can be demand-loaded by double-clicking on any icon representing it; if the **Options:ConfirmLoadofUnits** option is selected, a confirmation dialog box precedes loading.

The Units dialog box displayed by the **Browse:Units** command provides a convenient means of determining the loaded/unloaded status of every controlled unit in the current model.

Write-Protecting Controlled Units

Rose prevents you from modifying a write-protected controlled unit. Commands that load a controlled unit's model file—**File:Open** and **File:Units:Load**—will write-protect that unit if its model file's access control in the platform filesystem is read-only.

If a controlled unit is write-protected, the components or diagrams it contains are also write-protected unless they are nested controlled units. A nested controlled unit maintains write-protection independently of its parent.

Write-protection is enforced by removing the toolbar from write-protected diagrams, disabling certain menu items when the selected icon represents a write-protected component, and disabling the **OK** button in the specifications of write-protected components. A unit's write-protection status can be examined in the Units dialog box displayed by the **Browse:Units** command.

Rose's write-protection mechanism extends the exclusion mechanism used by platform configuration management Systems to synchronize the work of multiple users. Model files representing checked-in units have read-only access control in the filesystem; Rose allows a user to load a checked-in unit, but prevents modification of any component or diagram in the unit.

A unit's write-protection can be manually enabled or disabled using the **File:Unit:WriteProtection** command or the **Write Protect** and **Write Enable** buttons in the Units dialog box. These facilities allow you to load a checked-in unit, modify it, and save the result to a new model file to which you have write access. Alternatively, you may have loaded a checked-out unit with the intention of browsing rather than modification; manually write-protecting the unit assures that you will not inadvertently change it. The effects of manually changing a unit's write protection in this way persist only for the duration of your current Rose session, until you re-load the unit, or until you open a new model.

Note: *Manually changing a controlled unit's write-protection does not alter its model file's access control in the platform file system. If you load a checked-in unit and manually make it write-enabled, you will not be able to update the unit's model file without changing the model file's access control. Conversely, the* **File:Save** *and* **File:Units:Save** *commands (and their* **SaveAs** *variants) will attempt to update a write-protected unit's model file; success will depend entirely on the model file's access control in the platform filesystem.*

Managing Controlled Units

To designate a package as a controlled unit:

1. Select an icon representing the package.
2. Execute the **File:Units:Control** command.
3. Using the Filename for Unit dialog box, specify the name and location of a model file which will provide persistent storage for this controlled unit.

Rose designates the package as a controlled unit, and associates the filename you specified with this unit. The controlled unit is both loaded and write-enabled after this operation.

To unload a loaded controlled unit:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:Unload** command.

Rose unloads the controlled unit.

To load an unloaded unit:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:Load** command.
3. Using the Load From dialog box, select the model file from which to load the controlled unit; this dialog box presents the model file currently associated with this controlled unit as the default selection.

Rose associates the controlled unit with the selected file. If the platform filesystem's access control for the selected model file was read-only, then the controlled unit is write-protected; otherwise, the controlled unit is write-enabled.

To load a unit which is currently loaded—i.e. to update it with a new version:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:Reload** command.
3. Using the Load From dialog box, select the model file from which to load the controlled unit; this dialog box presents the model file currently associated with this controlled unit as the default selection.

Rose associates the controlled unit with the selected file. If the platform filesystem's access control for the selected model file was read-only, then the controlled unit is write-protected; otherwise, the controlled unit is write-enabled.

To save a controlled unit to its associated model file:

1. Select an icon representing the controlled unit.
2. Execute the **File:Save** command.

This operation will fail if the platform filesystem's access control setting for the model file is read-only.

To save a controlled unit to a specified model file:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:SaveAs** command.

Using the Save To dialog box, select the model file in which to save the controlled unit; this dialog box presents the model file currently associated with this controlled unit as the default selection.

This operation will fail if the platform filesystem's access control setting for the model file is read-only.

To designate a controlled unit as no longer controlled:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:Uncontrol** command.

The model file previously associated with the controlled unit is not deleted or modified by this command.

To prevent modification of a write-enabled controlled unit:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:WriteProtect** command.

The **File:Units:WriteProtect** command only appears on the **File:Units** menu if the icon you select represents a write-enabled controlled unit. This operation has no affect on the platform filesystem's access control setting for the model file associated with the selected controlled unit.

To allow modification of a write-protected controlled unit:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:WriteEnable** command.

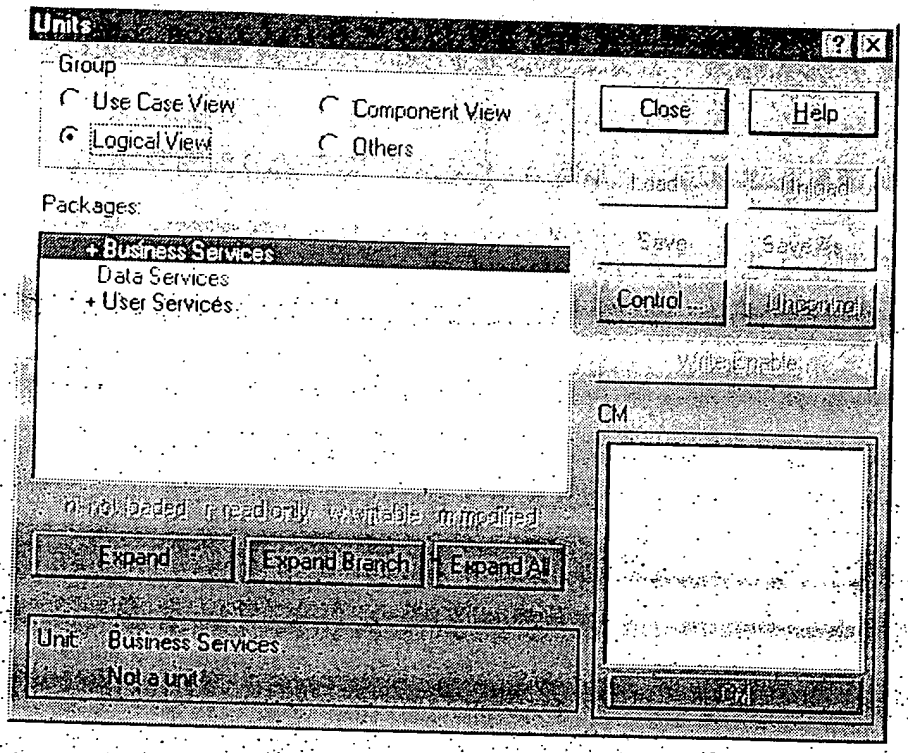
The **File:Units:WriteEnable** command only appears on the **File:Units** menu if the icon you select represents a write-protected controlled unit. This operation has no affect on the platform filesystem's access control setting for the model file associated with the selected controlled unit.

To invoke a configuration management command with a controlled unit as an argument:

1. Select an icon representing the controlled unit.
2. Execute the **File:Units:CM** command and select the desired configuration management from the displayed submenu.

Browsing Controlled Units

The Units dialog box, displayed by the **Browse:Units** command, allows you to both display and control the status of the model kernel, the deployment diagram, the code generation properties, each logical package, and each component package:



The **Group** option buttons enable you to choose which set of model components are displayed:

Use case view	Use cases in the model.
Logical view	All logical packages in the model.
Component view	All component packages in the model.
Others	The model kernel, deployment diagram, and code generation properties.

The display lists each model component of the kind designated by the **Group** option buttons, using indentation to indicate nesting. Each component's state is indicated by one or more letters to the left of its name:

n	The model component is not loaded.
r	The model component is write-protected.
w	The model component is write-enabled.
m	The model component has been modified after it was last saved.

You can select a model component by clicking on its display entry.

The **Collapse** and **Expand** buttons enable you to control the display of nested model components.

The **Unit** text box displays the name of the currently-selected model component.

If the selected model component is a controlled unit, the **File** text box displays the pathname of the file currently associated with this controlled unit.

The command buttons on the right side of the Units dialog box allow you to apply operations equivalent to those in **File:Units** to the selected model component.

Comparing Controlled Units

You can display the differences between two versions of a controlled unit by using Rose's Model Differencing tool, described in the *Model Differencing* chapter of this book.

The Path Map Mechanism

Rose provides the path map mechanism to support parallel development by teams of analysts, architects, and engineers. The path map enables Rose to create model files whose embedded pathnames are relative to a user-defined symbol. This allows Rose to work with models moved or copied among workspaces and archives by redefining the actual directory associated with the user-defined symbol.

The path map contains a list of entries; each entry represents a mapping between a virtual path symbol and an actual pathname. In the example below, the virtual path symbol is **$root**, and the actual pathname is **c:\user\trw\analyzer**:

```
root = c:\user\trw\analyzer
```

Note: *The leading dollar-sign must be omitted in the* virtual *path symbol definition, but must appear in all other usage.*

When a controlled unit is made or saved, each physical pathname embedded in its model file will be transformed to a virtual pathname by repeating the following steps until no replacement is made:

1. The head of the physical pathname is matched against the actual pathnames of the path map entries. If a match is found, the head is replaced by the virtual path symbol associated with the matching actual pathname. If more than one match is possible, the longest matching actual pathname is replaced by its associated virtual path symbol. To be a match, an actual pathname must be identical with a prefix of the physical pathname, segment by segment; all segments must match in their entirety.
2. If no match is found for the head of the physical pathname, the above matching algorithm is applied to the physical pathname starting with the second segment of the physical pathname rather than the first. If that fails, a match is attempted against the third segment and then the fourth and so on until either a match is found or all segments the physical pathname have been tested. When matching interior segments of the physical pathname, any drive name preceding the colon in the actual pathname is ignored.

A virtual pathname is created by replacing an actual pathname within a physical pathname with a virtual path symbol. Using the example above, the physical pathname **c:\user\trw\analyzer\source\parser.h** would be transformed to the virtual pathname **$root\source\parser.h**.

Rose does not convert physical pathnames in code generation properties to virtual pathnames; instead, you make such

pathnames virtual by constructing them with a virtual path symbol from your path map.

Virtual path symbols can be applied to four C++ Generator properties: model directory, component package directory, component spec filename and component body filename. Any time the path map is edited and you regenerate code, the virtual symbols in the C++ Generator properties are actualized with the current path map entries.

Traversal attributes may also contain virtual path symbols. In this case, the path map entries are dynamic. The effect of editing the path map through the path map editor is immediately reflected in the traversal attribute. For example, suppose you have the symbol:

project=d:\user\ken\rose

and you generate a class "company" to this directory. The path stored in the traversal attribute is:

$project\company.h

Now suppose you edit the path map using the path map editor such that:

project=d:\user\ken\codegen

If you try to reference "company.h" without saving the model or regenerating the code with the C++ Generator, Rose will expect to find company.h in:

d:\user\ken\codegen\company.h

and log an error.

When Rose opens, loads, imports, browses to source code, merges a model file, or uses a pathname specified in a code generation property, each virtual pathname is transformed back into a physical pathname using the path map mechanism. If your .ini file contained the following path map entry:

```
root = c:\user\ken\analyzer
```

then the virtual pathname **$root\source\parser.h** would be transformed to the physical pathname **c:\user\ken\analyzer\source\parser.h**.

The actual pathname in a path map entry can contain previously defined virtual symbols:

code $root/source

Virtual Pathmap Entry Extension

There are two extensions that expand on the functionality of virtual pathmap entries: a leading "&" and the "*" wildcard character.

A leading & on a pathname indicates the pathname is relative to the referencing context. If no referencing context exists, it is relative to the location of the model (**.mdl file**). This helps in referencing files in other directories that retain the same version number as that of the file that is referencing it.

Suppose a model

/rose/source/green.ss/sun30xyz.wrk/m.mdl

references a package

/rose/source/red.ss/sun30xyz.wrk/c.cat.

The **rose.ini** file may contain the following path map entry:

red=&/Links/red/rose/source/red.ss/sun27xyz.wrk/c.cat

When the pathmap entry is used to produce a virtual path for this reference, it expands to the actual path:

/rose/source/green.ss/sun30xyz.wrk/Links/red/rose/source/red.ss/sun27xyz.wrk/c.cat

which means the version of c.cat is in the same tower as /m.mdl.

The virtual name of

/rose/source/red.ss/sun30xyz.wrk/c.cat

is

$red/c.cat.

The second extension is the addition of "*" to wildcard a path segment. Suppose the **rose.ini** file contains the path map entry:

links = &Links/*/../..

This entry causes the pathmap variable links to be parameterized with the expansion of *. In the example above, the virtual name of c.cat when referenced from m.mdl becomes $links(red)/c.cat, which expands to the same actual path as before.

On startup and on execution of the **File:Open** command, Rose loads its path map from the `[Virtual Path Map]` section of the current user's .ini file. Entries should be of the form virtual path symbol = actual pathname, for example:

```
[Virtual Path Map]
mywork=c:\project\drs\workspace
```

If you change the **[Virtual Path Map]** in your **rose.ini** file, and are running a Rose release prior to 2.5.20, you must restart Rose in order to update its path map. If you are running release 2.5.20 or later, Rose updates its path map from **rose.ini** whenever the **Open** command executed.

Integrating with a Configuration Management System

Integration with the CM system is done through the customizable menus mechanism. If there is a customizable menu named **CM**, you will be able to access it through the **Units Dialog** box. A customizable menu file for PVCS is supplied with the product.

We recommend that the **CM** menu be located under the **Units** submenu of the **File** menu, but Rose does not enforce this choice.

Form PTO 948 (Rev. 8-98) U.S. DEPARTMENT OF COMMERCE - Patent and Trademark Office Application No. 09/204,973

NOTICE OF DRAFTSPERSON'S PATENT DRAWING REVIEW

The drawing(s) filed (insert date) 3/15/99 are:

A. ☒ approved by the Draftsperson under 37 CFR 1.84 or 1.152.

B. ☐ objected to by the Draftsperson under 37 CFR 1.84 or 1.152 for the reasons indicated below. The Examiner will require submission of new, corrected drawings when necessary. Corrected drawing must be sumitted according to the instructions on the back of this notice.

1. DRAWINGS. 37 CFR 1.84(a): Acceptable categories of drawings:
   Black ink. Color.
   ___ Color drawings are not acceptable until petiton is granted. Fig(s) _____
   ___ Pencil and non-black ink not permitted. Fig(s) _____
2. PHOTOGRAPHS. 37 CFR 1.84 (b)
   ___ 1 full-tone set is required. Fig(s) _____
   ___ Photographs not properly mounted (must use brystol board or photographic double-weight paper). Fig(s) _____
   ___ Poor quality (half-tone). Fig(s) _____
3. TYPE OF PAPER. 37 CFR 1.84(e)
   ___ Paper not flexible, strong, white, and durable. Fig(s) _____
   ___ Erasures, alterations, overwritings, interlineations, folds, copy machine marks not accepted. Fig(s) _____
   ___ Mylar, velum paper is not acceptable (too thin). Fig(s) _____
4. SIZE OF PAPER. 37 CFR 1.84(f): Acceptable sizes:
   ___ 21.0 cm by 29.7 cm (DIN size A4)
   ___ 21.6 cm by 27.9 cm (8 1/2 x 11 inches)
   ___ All drawing sheets not the same size. Sheet(s) _____
   ___ Drawings sheets not an acceptable size. Fig(s) _____
5. MARGINS. 37 CFR 1.84(g): Acceptable margins:
   Top 2.5 cm Left 2.5cm Right 1.5 cm Bottom 1.0 cm
   SIZE: A4 Size
   Top 2.5 cm Left 2.5 cm Right 1.5 cm Bottom 1.0 cm
   SIZE: 8 1/2 x 11
   Margins not acceptable. Fig(s) _____
   _____ Top (T) _____ Left (L)
   _____ Right (R) _____ Bottom (B)
6. VIEWS. 37 CFR 1.84(h)
   REMINDER: Specification may require revision to correspond to drawing changes.
   Partial views. 37 CFR 1.84(h)(2)
   ___ Brackets needed to show figure as one entity. Fig(s) _____
   ___ Views not labeled separately or properly. Fig(s) _____
   ___ Enlarged view not labeled separetely or properly. Fig(s) _____
7. SECTIONAL VIEWS. 37 CFR 1.84 (h)(3)
   ___ Hatching not indicated for sectional portions of an object. Fig(s) _____
   ___ Sectional designation should be noted with Arabic or Roman numbers. Fig(s) _____
8. ARRANGEMENT OF VIEWS. 37 CFR 1.84(i)
   ___ Words do not appear on a horizontal, left-to-right fashion when page is either upright or turned so that the top becomes the right side, except for graphs. Fig(s) _____
9. SCALE. 37 CFR 1.84(k)
   ___ Scale not large enough to show mechanism without crowding when drawing is reduced in size to two-thirds in reproduction. Fig(s) _____
10. CHARACTER OF LINES, NUMBERS, & LETTERS. 37 CFR 1.84(l)
    ___ Lines, numbers & letters not uniformly thick and well defined, clean, durable, and black (poor line quality). Fig(s) _____
11. SHADING. 37 CFR 1.84(m)
    ___ Solid black areas pale. Fig(s) _____
    ___ Solid black shading not permitted. Fig(s) _____
    ___ Shade lines, pale, rough and blurred. Fig(s) _____
12. NUMBERS, LETTERS, & REFERENCE CHARACTERS. 37 CFR 1.84(p)
    ___ Numbers and reference characters not plain and legible. Fig(s) _____
    ___ Figure legends are poor. Fig(s) _____
    ___ Numbers and reference characters not oriented in the same direction as the view. 37 CFR 1.84(p)(1) Fig(s) _____
    ___ English alphabet not used. 37 CFR 1.84(p)(2) Figs _____
    ___ Numbers, letters and reference characters must be at least .32 cm (1/8 inch) in height. 37 CFR 1.84(p)(3) Fig(s) _____
13. LEAD LINES. 37 CFR 1.84(q)
    ___ Lead lines cross each other. Fig(s) _____
    ___ Lead lines missing. Fig(s) _____
14. NUMBERING OF SHEETS OF DRAWINGS. 37 CFR 1.84(t)
    ___ Sheets not numbered consecutively, and in Arabic numerals beginning with number 1. Sheet(s) _____
15. NUMBERING OF VIEWS. 37 CFR 1.84(u)
    ___ Views not numbered consecutively, and in Arabic numerals, beginning with number 1. Fig(s) _____
16. CORRECTIONS. 37 CFR 1.84(w)
    ___ Corrections not made from prior PTO-948 dated _____
17. DESIGN DRAWINGS. 37 CFR 1.152
    ___ Surface shading shown not appropriate. Fig(s) _____
    ___ Solid black shading not used for color contrast. Fig(s) _____

COMMENTS

REVIEWER Q.D DATE 10/4/00 TELEPHONE NO. _____

ATTACHMENT TO PAPER NO. 4

Attachment for PTO-948 (Rev. 03/01, or earlier)
6/18/01

The below text replaces the pre-printed text under the heading, "Information on How to Effect Drawing Changes," on the back of the PTO-948 (Rev. 03/01, or earlier) form.

INFORMATION ON HOW TO EFFECT DRAWING CHANGES

1. Correction of Informalities -- 37 CFR 1.85

New corrected drawings must be filed with the changes incorporated therein. Identifying indicia, if provided, should include the title of the invention, inventor's name, and application number, or docket number (if any) if an application number has not been assigned to the application. If this information is provided, it must be placed on the front of each sheet and centered within the top margin. If corrected drawings are required in a Notice of Allowability (PTOL-37), the new drawings **MUST** be filed within the **THREE MONTH** shortened statutory period set for reply in the Notice of Allowability. Extensions of time may NOT be obtained under the provisions of 37 CFR 1.136(a) or (b) for filing the corrected drawings after the mailing of a Notice of Allowability. The drawings should be filed as a separate paper with a transmittal letter addressed to the Official Draftsperson.

2. Corrections other than Informalities Noted by Draftsperson on form PTO-948.

All changes to the drawings, other than informalities noted by the Draftsperson, **MUST** be made in the same manner as above except that, normally, a highlighted (preferably red ink) sketch of the changes to be incorporated into the new drawings **MUST** be approved by the examiner before the application will be allowed. No changes will be permitted to be made, other than correction of informalities, unless the examiner has approved the proposed changes.

Timing of Corrections

Applicant is required to submit the drawing corrections within the time period set in the attached Office communication. See 37 CFR 1.85(a).

Failure to take corrective action within the set period will result in **ABANDONMENT** of the application.

Interview Summary

Application No.	Applicant(s)
09/204,973	David Lars Ehnebuske et al.
Examiner	**Group Art Unit**
Todd Ingberg	2122

All participants (applicant, applicant's representative, PTO personnel):

(1) *Todd Ingberg* (3) _____

(2) *Stephen Walder JR. (41,534)* (4) _____

Date of Interview *Jun 13, 2001*

Type: a)☒ Telephonic b)☐ Video Conference
c)☐ Personal [copy is given to 1)☐ applicant 2)☐ applicant's representative]

Exhibit shown or demonstration conducted: d)☐ Yes e)☒ No. If yes, brief description:

Claim(s) discussed: *1-98*

Identification of prior art discussed:

Agreement with respect to the claims f)☐ was reached. g)☒ was not reached. h)☐ N/A.

Substance of Interview including description of the general nature of what was agreed to if an agreement was reached, or any other comments:

Examiner pointed out the claims (32 pages) are redundent and the limitations fail to further distinguish the invention from the Specification (22 pages). Applicant feels the scope of the claims differ. Examiner interprets the claimed invention as a candidate for Restriction based on Multiplicity.

(A fuller description, if necessary, and a copy of the amendments which the examiner agreed would render the claims allowable, if available, must be attached. Also, where no copy of the amendments that would render the claims allowable is available, a summary thereof must be attached.)

i)☒ It is not necessary for applicant to provide a separate record of the substance of the interview (if box is checked).

Unless the paragraph above has been checked, THE FORMAL WRITTEN REPLY TO THE LAST OFFICE ACTION MUST INCLUDE THE SUBSTANCE OF THE INTERVIEW. (See MPEP section 713.04). If a reply to the last Office action has already been filed, APPLICANT IS GIVEN ONE MONTH FROM THIS INTERVIEW DATE TO FILE A STATEMENT OF THE SUBSTANCE OF THE INTERVIEW. See Summary of Record of Interview requirements on reverse side or on attached

Examiner Note: You must sign this form unless it is an Attachment to a signed Office action.